

UNCLASSIFIED

AD NUMBER: ADB340643

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

Distribution authorized to U.S. Government agencies only;
Administrative/Operational Use; Jun 2008. Other requests shall be
referred to Air Force Research Laboratory, ATTN: RITB, Rome, NY
13441-4505

AUTHORITY

AFRL memo dtd 12 Jun 2019

THIS PAGE IS UNCLASSIFIED



DEPARTMENT OF THE AIR FORCE
AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE ROME NY

12 June 2019

MEMORANDUM FOR DEFENSE TECHNICAL INFORMATION CENTER
ATTN: MR. ROBERT STOKES
DTIC-CQ
8725 JOHN J. KINGMAN ROAD
FORT BELVOIR VA 22060

FROM: AFRL/RI STINFO
26 Electronic Parkway
Rome, NY 13441-4514

SUBJECT: Reclassification of Technical Report AFRL-RI-RS-TR-2008-160, "R-STREAM 3.0 COMPILER", June 2008.

1. AFRL/RITB requests reclassification of subject document from Distribution B, U.S. Government Agencies Only to Distribution A, Public Release.
2. The document was initially classified as distribution B to protect technical or operational data or information from automatic dissemination and to furthermore protect potentially patentable information from premature dissemination.
3. Since the publication of the report in 2008, the technical or operational data or information has been sensibly disseminated and the technology is also now protected commercially by patent.
4. Please direct any questions to Mr. Christopher Flynn, AFRL/RITA, Christopher.Flynn.6@us.af.mil, 315-330-3249.

A handwritten signature in black ink, appearing to read "Emily Wager", is positioned above the printed name.

EMILY WAGER
AFRL/RI STINFO Program Manager

Attachment:
Technical Report AFRL-RI-RS-TR-2008-160

Cc: AFRL/RITA (C. Flynn)

AFRL-RI-RS-TR-2008-160
Final Technical Report
June 2008



R-STREAM 3.0 COMPILER

Reservoir Labs, Inc.

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. P370

DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES ONLY; ADMINISTRATIVE OR OPERATIONAL USE; JUN 08. OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO AFRL/RITB, ROME, NY 13441-4505.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

DESTRUCTION NOTICE - For classified documents, follow the procedures in DOD 5220.22-M, National Industrial Security Manual (NISPOM), section 5-705 or DOD 5200.1-R, Information Security Program, Chapter VI. For unclassified limited documents, destroy by any method that will prevent disclosure of contents or reconstruction of the document.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

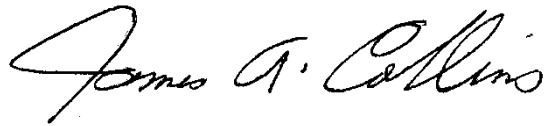
Qualified requestors may obtain copies of this report from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2008-160 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:



CHRISTOPHER FLYNN
Work Unit Manager



JAMES A. COLLINS, Deputy Chief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE*Form Approved*
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) JUN 08		2. REPORT TYPE Final		3. DATES COVERED (From - To) Apr 03 – Dec 07	
4. TITLE AND SUBTITLE R-STREAM 3.0 COMPILER				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER F30602-03-C-0033	
				5c. PROGRAM ELEMENT NUMBER 62712E	
6. AUTHOR(S) Richard Lethin, Allen Leung, Benoit Meister, Peter Szilagyi, Nicholas Vasilache and David Wohlford				5d. PROJECT NUMBER P370	
				5e. TASK NUMBER HL	
				5f. WORK UNIT NUMBER CM	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Reservoir Labs, Inc. 632 Broadway, Ste 803 New York, NY 10012				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/RITB 3701 North Fairfax Drive 525 Brooks Rd Arlington VA 22203-1714 Rome NY 13441-4505				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2008-160	
12. DISTRIBUTION AVAILABILITY STATEMENT <i>DISTRIBUTION AUTHORIZED TO U.S. GOVERNMENT AGENCIES ONLY; ADMINISTRATIVE OR OPERATIONAL USE; JUN 08. OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO AFRL/RITB, ROME, NY 13441-4505.</i>					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT This report describes Reservoir Lab's R-Stream compiler and mapper component developed in DARPA's Polymorphous Computing Architecture (PCA) program. PCAs are typically multi-core distributed memory machines without coherent global memory. R-Stream is a source-to-source compiler. As such, it acts as a High-Level Compiler (HLC), accepting C programs with user-selected mappable regions as input, and produces parallelized and mapped C programs as output. R-Stream is the result of a research effort to attack the problem of automatic mapping to these important hardware attributes directly and rigorously, using the most advanced theory in automatic high-level optimizing available, to provide a robust and extendable implementation, and to advance the state of the theory and practice in high-level optimization.					
15. SUBJECT TERMS Polymorphic, Compiler					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT SAR	18. NUMBER OF PAGES 186	19a. NAME OF RESPONSIBLE PERSON Christopher Flynn
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) 315-330-3249

Contents

1	Introduction	1
1.1	Obtaining R-Stream	2
1.2	Automatic Mapping	2
1.2.1	Parallelism extraction and enhancement	3
1.2.2	Locality optimizations	5
1.2.3	Iteration space tiling	5
1.3	Computation and data distribution	6
1.4	Data layout optimizations	6
1.4.1	Bulk communication generation (a.k.a. DMA generation)	7
1.5	Limitations of R-Stream	7
1.6	Organization	9
2	The Polyhedral Model	10
2.1	Polyhedra and polytopes	10
2.1.1	Parameters	10
2.1.2	Z-polyhedra	11
2.1.3	Domain and Z-domain	11
2.2	Modeling iteration spaces	11
2.3	Modeling dependences	12
2.4	Space-time mappings	13
2.5	Example	13
2.6	Further Readings	15
3	Mapper Architecture	16
4	The Generalized Dependence Graph	20
4.1	Space-time mapping	20
4.2	Dependence edges	22
4.3	Example 1	22
4.4	Example 2	23
4.5	Related Works	26
5	Polyhedral Mapper Infrastructure	27
6	Array Expansion	30
6.1	Related works	31
6.2	Current algorithm	31
6.2.1	Example	32
7	Affine Scheduling	34
7.1	General template of affine scheduling algorithms	34
7.2	Feautrier's algorithm	35
7.3	Darte and Vivien's algorithm	35
7.4	Lim and Lam's affine partitioning algorithm	37
7.5	Summary	37

7.6	Our Algorithm	38
7.6.1	Computing “wavy” schedules	39
7.7	Summary and Related Works	39
8	Forming Kernels: Grouping and Tiling	41
8.1	Grouping	44
8.2	Tiling	46
8.2.1	Orthogonal tiling	46
8.2.2	Constraints derived from the target architecture	46
8.3	Formulation of the tiling problem	47
8.3.1	Hoisting permutable loops	47
8.3.2	Loop sinking instead of hoisting	49
8.3.3	Tilability	51
8.3.4	Tiling as a search: beta tree	53
8.3.5	Consequences of our tiling paradigm	54
8.3.6	Implementation: generic search	55
8.4	Interaction with other mapper components	55
8.5	Future improvements	56
9	Processor Placement	57
9.1	Algorithm	57
9.2	Single-Program Multiple-Data (SPMD) code generation	58
9.3	Minimizing communications	59
9.4	Eliminating host broadcasts	59
9.5	Related works	59
10	Local Memory Compaction	61
10.1	Motivating Examples	61
10.2	Algorithm	64
10.3	Group related references	65
10.4	Hermite Decomposition	66
10.4.1	Example 1	67
10.4.2	Example 2	67
10.5	Unimodular Reindexing	68
10.5.1	Solving the optimization problem	69
10.6	Generating bulk communication	70
10.7	Related Work	72
11	Multi-buffering	74
11.1	Multi-buffering with loop interchange	74
11.2	Multi-buffering with loop jamming	77
11.2.1	Shifting problem	79
11.2.2	New multi-buffering scheme: summary	83
11.3	Hierarchical multi-buffering	84

12 DMA Optimization	86
12.1 Example	87
12.2 Algorithm	89
12.3 Special cases	90
12.3.1 Big packets	90
12.3.2 Strides not allowed on one side	90
12.3.3 Strides not allowed on any sides	91
12.3.4 Bijection between both sides	91
12.4 Further optimization	91
12.4.1 Simplifying the data transfers by transferring more	91
12.4.2 Optimizing for data transfer size	92
12.4.3 Optimizing for memory banks	92
12.5 Implementation	93
13 Register Tiling	94
13.1 Implementation	94
14 Array Contraction	95
14.1 Lattice based framework	95
14.2 Algorithm	96
14.3 Example	97
15 Polyhedral Scanning	99
15.1 Example	99
15.2 Formal statement	100
15.3 Related works	101
15.4 R-Stream’s polyhedral scanner	101
15.5 Performance improvements	103
15.6 Code quality improvements	105
15.6.1 Controlling domain splitting	106
15.6.2 Constraints tightening	108
15.6.3 Predicate and stride hoisting	109
15.6.4 Controlling code duplication	111
16 The R-Stream Compiler Infrastructure	114
16.1 The Sprig IR	114
16.1.1 Heterogeneous compilation	115
16.1.2 Source level type system	115
16.1.3 Source code regeneration	116
16.2 Scalar optimizations and analyses	116
17 Raising: IR to Polyhedral Form	117
17.1 Raising algorithm	117
17.1.1 Pointer analysis	118
17.1.2 Mappable region identification	119
17.1.3 Inlining	120

17.1.4	Index, data and predicate values classification	120
17.1.5	If-conversion	121
17.1.6	Statement formation	122
17.1.7	Base address and parameters detection	123
17.1.8	Recurrence analysis	123
17.1.9	GDG building	126
17.2	Future extensions	127
17.2.1	Automatic region selection and inlining	127
17.2.2	Abstract data types via struct and unions arguments . . .	127
17.2.3	Heap memory management and array delinearization . . .	127
17.2.4	Geometric recurrences	129
17.2.5	Modulo recurrences	129
18	Lowering: From Polyhedral Form to Target Code	131
18.1	Syntax reconstruction	131
18.2	Algorithm	132
19	CELL Backend	135
19.1	Local memory and DMA	135
19.2	CELL target API	135
19.3	DMA primitives	136
19.4	Memory primitives	137
19.5	Synchronization primitives	137
19.6	Cell mapping example	138
19.6.1	Manual SIMDization	141
20	TRIPS Backend	144
20.1	TRIPS target API	144
20.2	Mapping example	145
21	SMP Backend	152
22	R-Stream and Polymorphous Computer Architectures	153
22.1	PCA program objectives	153
22.2	PCA hardware strategy	154
22.3	PCA software strategy	155
22.3.1	Need for definition of “mapping”	156
22.3.2	Power and limitations of polyhedral compilers	157
22.3.3	Need for a uniform abstraction, phase fusion	158
22.3.4	Power efficiency	158
22.3.5	Dynamic Morphing	159
22.4	Transitions	160
22.5	Summary	161

List of Figures

1	Mapper Architecture.	16
2	Polyhedral mapper software infrastructure.	27
3	Uniformization of a dependence edge.	36
4	Tiling viewed as iteration collapsing.	50
5	Tilability propagated on a β -tree.	53
6	Hierarchical multi-buffering.	85
7	Quill�re’s algorithm.	102
8	Improved separation algorithm.	104
9	The R-Stream infrastructure.	114
10	Pointer analysis example	118
11	Object summary graph	119
12	Control structure reconstruction.	132
13	Expression building.	133
14	Idiom matching example.	134

List of Tables

1	Summary of scheduling algorithms.	37
2	CLooG versus Bungle (64-bits).	105
3	Idiom matching.	134
4	Performance of matrix multiply on PS3.	142

Glossary

The following terms and acronyms are used frequently across this report. Since they are not defined for each section in which they are used, here we provide a definition accessible at any time.

Cell : Heterogeneous chip co-developed by Sony, Toshiba and IBM (STI) [MD05a], which contains a Power PC processor, called PPU (for *Power Processing Unit*) and eight SIMD processing elements called SPUs (for *Synergistic Processing Unit*) on the same chip.

FLOPS : FLOating point OPerations per Second. Standard measure of performance that indicates how many floating-point operations are executed by a computer per second. For a given target machine, the FLOPS can be measured as an average over the execution of a given algorithm or as an absolute achievable value.

FPGA : Field Programmable Gate Array. Processor made of a high number of configurable logic blocks, which can perform a number of different functions depending on their configuration. The way the blocks are connected together is also configurable. The configuration for a whole FPGA is encoded as a bit stream.

GDG : Generalized Dependence Graph. This is how the part of the program that is to be mapped is represented in R-Stream. It is a graph whose vertices represent program statements and whose edges represent inter-statement dependences.

HLC : High-level compiler. In the compilation scheme introduced by the Morphware forum, the high-level compiler is responsible for producing a set of sequential programs, each of which is to be executed on one processing element of the target machine, after being compiled by the low-level compiler.

HPC : High Performance Computing - supercomputing

HPEC : High Performance Embedded Computing. Embedded computing is computing that is situated between sensors and/or actuators. High Performance Embedded Computing is distinguished by being supercomputing rates of performance associated with sensors, e.g., as in advanced radars.

IR : Intermediate representation. In compilers, this refers to any internal representation of the program. It is an intermediate form between the input program (the text of the source) and the output program (either the binary executable code or the text of the target code when the compiler is source-to-source). There may be several IRs in a compiler. In R-Stream, while several representations are used, only one representation is named “the IR”. It is the operator graph based representation called “Sprig”, described in Section 16.1.

- LLC : Low-level compiler. In the compilation scheme introduced by the Morphware forum, the low-level compiler is responsible for compiling programs for a single processing element. Such a program can be the output of the High-Level Compiler.
- PCA : Polymorphous Computer Architecture. A class of computer architecture that achieves high computational efficiency (FLOPS/W) on a broad class of applications and that is programmable. The term was coined in the 2001-2007 DARPA program of this name to produce instances of these architectures, and the programming tools for them. These architectures achieve their versatility through the use of parallelism, distributed local memories for data and instructions, explicit control of communication and memory, SIMD aspect, tiling and replication of functional units. Commercial architectures have appeared which exhibit these features, such as Cell.
- PE : Processing element. A processor that is part of a set of processors across which R-Stream is expected to distribute computations of an input program. In the current machine model, such a set is organized as a grid, i.e., a dense hyper-rectangular set.
- SIMD : Single-Instruction, Multiple Data. Instruction set and – by extension – processor that executes the same instruction on several data at once. Many modern processors have SIMD capabilities.
- SIMDization : Program transformation that attempts to exploit the SIMD features of a processor.
- SMP : Symmetric Multi-Processor. A homogeneous grid of processing elements in a shared memory machine.
- SWEPT : Size Weight Energy Power Time. These are metrics of performance, beyond simply the execution time, that are particularly relevant in the HPEC application domain, and rapidly becoming important in the HPC application domain.
- TRIPS : Stands for *Tera-op, Reliable, Intelligently adaptive Processing System*. TRIPS [BKM04, SNG⁺06] is a chip developed at the Computer Science Department of University of Texas as part of the DARPA PCA program. It carries a homogeneous grid of processing elements (called *tiles*) on a single chip.

1 Introduction

This report describes Reservoir Lab’s R-Stream compiler and mapper component developed in DARPA’s Polymorphous Computing Architecture (PCA) project.

Mapping in our context is the process of transforming sequential programs into efficient parallel code to be executed on PCAs, which are a new generation of high performance architectures with very high potential computational efficiency (FLOPS/W) yet which are programmable.

Automatic mapping is a critical technology for achieving the potential of these architectures, because these chips simultaneously achieve high FLOPS/W and programmability by otherwise compromising on the complexity of the programming task. The goal of automatic mapping is to make programmers productive: to broaden the set of programmers who can achieve the potential of the architectures beyond a few gurus who hand-code the application or limited libraries, and to make even more complex variants of the architectures usable by the gurus.

The application domain that we address is high performance scientific, signal and image processing. To further restrict the domain to make the mapping problem tractable, we will assume that the input programs are mainly *static control programs* that operate on dense matrices and arrays. This is the class of programs consisting of do-loops with loop bounds that are affine functions of outer indices and parameters, and array indexing functions that are affine functions of loop indices and parameters. Irregular data structures like sparse matrices are not considered. Even with those application domain restrictions, constructs outside of this model, such as data dependent conditionals and non-affine array indices, are handled conservatively.

PCAs are typically multi-core distributed memory machines without coherent global memory.¹ Thus in addition to partitioning the program and data into a distributed form, the mapper is also responsible for inserting explicit communication and synchronization between processing elements at strategic points in the mapped program. Exposed architectural features, such as Direct Memory Access (DMA) controllers, Single-Instruction-Multiple Data (SIMD) arithmetic engines, scratch-pad memory, hardware FIFOs and reconfigurable dataflow networks, also have to be explicitly managed by the mapping.

R-Stream is a source-to-source compiler. As such, it acts as a High-Level Compiler (HLC), accepting C programs with user-selected **mappable regions** as input, and produces parallelized and mapped C programs as output.² The output of R-Stream is the input program mapped to the target architecture, to the exposed architectural features of PCA targets. Executing the output code requires a Low-Level Compiler (LLC) to accept the mapped program, and to

¹Although some of our target architectures do have shared memory, it is less efficient than using memories local to the computation units, with communication and synchronization managed explicitly.

²Significant engineering is present in R-Stream so that the C output looks readable, preserves debuggability, and to generate idiomatic forms that work well with different Low-Level Compilers (LLCs). C is chosen for convenience. For some of our projects, e.g., our output stages can produce the result in languages other than C, e.g., in FPGA-specific languages.

generate code for the individual accelerator engines or host processors, using the widely and relatively generic compiler technologies for instruction selection, instruction scheduling, register allocation, and so forth.

R-Stream is the result of a research effort to attack the problem of automatic mapping to these important hardware attributes directly and rigorously, using the most advanced theory in automatic high level optimizing available, to provide a robust and extendable implementation, and to advance the state of the theory and practice in high level optimization.

1.1 Obtaining R-Stream

R-Stream is available now for use, research, and evaluation:

1. Government employees can obtain the source, binaries, and documentation, by contacting the cognizant program officer, Christopher Flynn of AFRL, (315) 330-3249, Christopher.Flynn@rl.af.mil, 25 Brooks Rd., Rome, NY 13441-4505. R-Stream is under active development, so Reservoir Labs will endeavor to provide updated distributions upon request directly to U.S. government departments, as well as support and customization as required. Reservoir Labs is interested in and able to apply and extend R-Stream for government programs with HPC and HPEC components. Government contractors and FFRDCs should contact Reservoir Labs directly.
2. For academic collaboration, Reservoir Labs has developed a model license for universities. In this license, Reservoir Labs provides source code to the university to enable their research in advanced compilers, programming languages, and high performance computing, based on R-Stream. Reservoir Labs can optionally be a research collaborator, and is available as a motivated commercial entity and channel for transitioning and proliferating the results of such university results into use. The license provides also for university commercialization of the results. Please contact Reservoir Labs directly for more information.
3. Finally, Reservoir Labs is executing projects of commercial transition of this research technology in a number of forms, including executables targeted to specific advanced processors. Reservoir Labs is providing customization and support, and is using R-Stream as the basis for a number of advanced research and development projects. Commercial entities with interest in using or applying R-Stream should contact Reservoir Labs directly.

1.2 Automatic Mapping

User-selected mappable regions are mainly computationally intensive program fragments in the form of *static control programs*, that is, imperfect loop nests operating on arrays where the loop bounds and array indices are affine functions

of the indices of enclosing loops and incoming parameters. Such program fragments constitute the main kernel or kernels of many existing high-performance computing (HPC) and high-performance embedded-computing (HPEC) applications.

The R-Stream mapper expects minimal user-provided hints are provided to direct the mapping process. In contrast to other programming paradigms, like for instance those of OpenMP and MPI, in which the programmer has to specify the parallelism, locality and/or the grain of communication he wants to exploit, R-Stream uses *automatic* program transformation techniques to derive the final mapped program. The mapper customizes the mapping process by referring to the machine model of each target architecture.

The novel automatic program transformation techniques form the core of the R-Stream mapper. These are described in the following subsections.

1.2.1 Parallelism extraction and enhancement

Unlike traditional vectorizing compilers, R-Stream is able to obtain both fine-grained (inner-loop) and coarse-grained (outer-loop) parallelism within the same framework. Exploiting parallelism at the coarse-grain level is necessary to amortize the startup cost of interprocessor communication, which is frequently non-trivial. Exploiting fine-grain parallelism is necessary to take advantage of instruction level parallelism (ILP), SIMD parallelism, or parallelism available in reconfigurable dataflow networks embedded in a PCA processor.

For example, the sequential loop:

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    A[i][j] = A[i-1][j+1];
```

can be transformed into a loop nest with fine-grained parallelism via loop reversal and interchange:

```
for (i = 1-M; i <= 0; i++)
  doall (j = 0; j < N; j++)
    A[j][-i] = A[j-1][1-i];
```

or restructured into a loop nest with coarse-grained parallelism via *loop skewing*, which consists in combining inner loop variables with affine combinations of outer loop variables:

```
doall (i = 0; i < N+M-1; i++)
  for (j = max(0, 1+i-M); j < min(N-1, i-1); j++)
    A[j][i-j] = A[j-1][i-j+1];
```

We approach the problem of parallelism extraction via *affine scheduling*, which generalizes and improves on classical loop transformation techniques.

Unlike the latter, affine scheduling is not restricted to perfectly nested loops, unimodular transformations, or single statement/single body loop nests.

For example, using *affine partitioning*, a variant of affine scheduling, the following imperfectly nested loops:³

```
for (i = 1; i < n; i++) {
  for (j = 0; j < n; j++) {
    for (k = 0; k < n - j; k++)
      a[i][j+k] = a[i][j+k] + a[i-1][j+k] * b[j][k];
    for (k = 1 + j; k < n; k++)
      a[i][k] = a[i][k] - a[i][j] * b[n-j-1][k];
  }
}
```

can be transformed into a loop nest with coarse-grained parallelism:

```
if (n >= 2)
  a[1][0] = a[1][0] + a[0][0] * b[0][0];
for (i = 2; i < n; i++) {
  doall (j = 1; j < i; j++) {
    for (k = j-1; k < 0; k++) {
      a[j][i-j] = a[j][i-j] + a[j-1][i-j] * b[i-j+k][-k];
      a[j][i-j] = a[j][i-j] - a[j][i-j+k] * b[-i+j-k+n-1][i-j];
    }
    a[j][i-j] = a[j][i-j] + a[j-1][i-j] * b[i-j][0];
  }
  a[i][0] = a[i][0] + a[i-1][0] * b[0][0];
}
for (i = n; i < 2*n-1; i++) {
  doall (j = 1+i-n; j < n; j++) {
    for (k = j-1; k < 0; k++) {
      a[j][i-j] = a[j][i-j] + a[j-1][i-j] * b[i-j+k][-k];
      a[j][i-j] = a[j][i-j] - a[j][i-j+k] * b[-i+j-k+n-1][i-j];
    }
    a[j][i-j] = a[j][i-j] + a[j-1][i-j] * b[i-j][0];
  }
}
```

The above loop nest can in principle be obtained from the original by applying a carefully guided sequence of loop reversal, interchange, skewing, reversal, and fusion in a traditional loop restructuring framework, although the current example is likely to exceed the capability of most implementations. With affine scheduling techniques, the same result can be obtained via much more direct means.

We shall describe our implementations of these techniques in Section 7.

³This example is taken from [LL97].

1.2.2 Locality optimizations

Locality is a critical criterion to efficient execution on multi-core and distributed memory architectures. Locality can be observed in various system levels, from registers to memory, from different on-chip cores to within a system as a whole.

Traditionally, we say that *temporal locality* is obtained when the same memory element is accessed within a small window of time during execution. Generally, *spatial locality* is obtained when elements of a contiguous *zone* of memory (that typically translates to a cache line, a memory page, a processing element's local memory, etc.) are accessed successively or within a short amount of time.

Consider two statements s_1 and s_2 . If iteration i_1 of statement s_1 accesses the same data element as iteration i_2 of statement s_2 , temporal locality of these accesses is achieved if i_1 and i_2 are executed successively or within a short amount of time. As there are usually many such instances of (i_1, i_2) , the loops must be restructured so that s_1 and s_2 are consecutive statements of a same loop and that i_1 is close (or equal) to i_2 . This restructuring transformation is known as *fusion* in the literature. See below:

```
                                // After fusion
for (i = ...)                  for (i = ...) {
    S1;                          S1;
for (i = ...)                  S2;
    S2;                          }
```

This is also true for a single statement: if two iterations of the statement access the same memory element, they should be executed consecutively or within a short amount of time. Indeed this can be seen as a particular case of fusion. In this case, particular array references may reuse a given memory element for a number of times that depends only on the loop bounds.

For example, the following loop nests on the left can be transformed into the loop nests on the right, which has better temporal locality.

```
                                // Better temporal locality
for (i = ...)                  for (j = ...)
    for (j = ...)              for (i = ...)
        r += A[j] * C[j];      r += A[j] * C[j];
```

1.2.3 Iteration space tiling

Spatial and temporal locality of a program can also be improved by *blocking* or *iteration space tiling* (or tiling for short.) A classical example of tiling is the transformation of a matrix multiply loop nests, shown below,

```
for (i = 0; i < 127; i++)
    for (j = 0; j < 127; j++)
```

```

for (k = 0; k < 127; k++)
    C[i][j] += A[i][k] * B[k][j];

```

into the following tiled loop nests:

```

for (i = 0; i <= 112; i += 16)
    for (j = 0; j <= 112; j += 16)
        for (k = 0; k <= 112; k += 16)
            for (l = i; l <= i + 15; l++)
                for (m = j; m <= j + 15; m++)
                    for (n = k; n <= k + 15; n++)
                        C[l][m] += A[l][n] * B[n][m];

```

The large matrices have been split into 16×16 submatrices. Because elements of these submatrices are repeatedly accessed together within the innermost (l,m,n) loop nests, space locality has been improved.

The R-Stream mapper also uses tiling to form parallel tasks to be executed on the grid of processing elements. In a shared memory system, each task can modify a limited amount of data (that hopefully fits in the cache) before processing another data set. In a distributed memory environment, each such task is meant to have its own local memory space. By choosing which statements constitute a tile and by adjusting the tile sizes, we can reduce the number of cache misses (in shared memory) or the amount of communication (in distributed memory). Our tiling algorithm will be presented in Section 8.

1.3 Computation and data distribution

Tasks formed in the tiling phase have to be distributed onto processors in such a way that they can be executed in parallel. This is called processor *placement*. Efficient placement strategies aim to reduce the amount of broadcasts and peer-to-peer communications – between the host and the processing elements and among the processing elements – and the distance between communicating processing elements.

The approach chosen in R-Stream follows the *computer owns* rule. Tasks are mapped onto the space of processors with the goal of improving locality and minimizing communications. After placement, the communications necessary for a given computation can then be derived.

Section 9 describes the R-Stream placement algorithm in details.

1.4 Data layout optimizations

Tasks that are mapped onto different processors have to communicate with each other and with the main host processor. When migrating data from one processor to another, we have a further opportunity to change the data layout to improve storage utilization and locality of references. For example, when inserting explicit communication into the loop

```
double A[300,300];
...
for (i = 0; i < 100; i++) {
    ... = ... A[2*i+100,3*i] ...;
}
```

we can utilize a different layout the array **A** in local memory:

```
double A_local[100];
...
Transfer data from A[2*i+100,3*i] to A_local[i]
for (i = 0; i < 100; i++) {
    ... = ... A_local[i] ...;
}
```

Transforming the reference from $A[2*i+100,3*i]$ to $A_local[i]$ reduces the storage requirements of local memory from 300×300 elements to 100 elements.

Section 10 will be devoted to our local memory compaction algorithms that perform this task.

1.4.1 Bulk communication generation (a.k.a. DMA generation)

Communication commands created in the mapping process have to be physically realized as actual operations supported by the underlying architecture, through either system calls (to DMA engines), library calls (such as the Message Passing Interface (MPI), or RASCLib [SGI07] in the case of SGI FPGA servers such as the RC100) or calls to an abstraction layer which is part of R-Stream (as in the case of Cell and TRIPS).

This is the job of the *bulk communication generation* (also referred to as *DMA optimization* for short, see Section 12) component of the mapper.

On some architectures like the Cell [MD05b, GHF⁺05, GHF⁺06] and TRIPS [BKM04, SNG⁺06], communication commands are mapped into operations to the DMA engines issued from the PE's. On other architectures, such as some FPGA systems, they are turned into buffer management and DMA operations to be issued on a host processor.

1.5 Limitations of R-Stream

As the product of a research project, the R-Stream has certain limitations. While it illustrates and provides a proof of concept for the use of polyhedral methods for optimization to PCAs, it is not yet a tool suitable for “production use.”

Some of the limitations arise from the polyhedral approach. The scope of applicability of the technique is currently limited to the class of programs that are structurally able to be raised to the polyhedral model, and which do not have expression level problems that prevent the raising phase from inferring important

properties, e.g., that arrays are abstract. We can extend the scope of applicability through improvements to the analyses that support raising, e.g., better alias analysis to determine abstractability of arrays, or privatization transformations for regions of code that would otherwise be unraisable. The structural scope can be extended through approximations; for example, we are working to extend the scope to include loops that include conditionals through the introduction of predication to the representation, and working this through the mapper.

The polyhedral approach also imposes limits based on the scalability of the underlying algorithms. The expanded scope and precision and transformations available in the polyhedral approach comes at a cost in the computational complexity of the optimizations. This limits the size of loop nests that we can currently optimize to the hundreds of lines. We are actively working on improving the scale and performance of the mapper through better algorithms and better implementations. We do not see any reason why one shouldn't be able to scale algorithms to be able to optimize loop nests with 10's of thousands of lines.

A further limitation to the polyhedral approach is that it is restricted to dependence preserving transformations. In many cases, the optimal transformation for a given application involves changes at a higher level, e.g., algorithmic changes. The simplest example of this is changing a reduction from a recurrence to being a parallel prefix - such a change involves exploiting the associativity of the arithmetic operators to rewrite the dependence graph of the computation. However, more complex transformations are often needed, such as implementing a QR decomposition algorithm using a Givens based algorithm instead of a Householder algorithm. **R-Stream** is based on the assumption that the user has chosen the right algorithm. In future research, we will explore higher level algorithm transformation frameworks. Existing frameworks in the literature often suffer from the inability to predict the performance of a given variant in their algorithm search. There is an opportunity in forward research to study the interaction of these searches with downstream optimizations using the polyhedral model.

Limitations in **R-Stream** arise from our particular choice and implementation of mapping phases. Choices made by upstream mapping phases are often guesses about what downstream phases will do. These guess can turn out to be wrong, reducing performance. We are currently working on improvements to the phase architecture of the mapper, working to combine the phases in some cases, or to tune the tradeoffs.

There are some bugs in the mapper that are due to use of approximations, that we are working to address. For example, in the communication generation phase, there are some "edge case" bugs where dependences might be violated in the output code, due to the use of approximations in the code representation. We have implemented more precise representations and are working to fix the mapper now.

In general, **R-Stream** needs more tuning and use. We need to better characterize the benefits and limitations of the mapper. Since this is a recent and leading edge optimization tool, little is known about the performance and limi-

tations of it.

Reservoir Labs is actively working on these limitations in internal and follow on research projects, and as we apply **R-Stream** to more architectures and application domains.

1.6 Organization

The rest of this report is organized as follows. Section 2 gives a brief introduction to the polyhedral model, the mathematical framework of the **R-Stream** mapper. The architecture of the mapper is presented in Section 3. Section 4 then describes the internal representation of the **R-Stream** mapper called the generalized dependence graph. Sections 6, 7, 8, 9, 10, 11, 12, 13 and 14 then describe the inner workings of various mapper optimizations and transformations in details. In Section 15, we present the code generation algorithm of the mapper.

In Section 16, we take a longer view by examining the entire **R-Stream** compiler infrastructure and describing how the mapper fits into the overall framework. In Sections 17 and 18 we present the *raising* and *lowering* algorithms, i.e., the conversion algorithms the **R-Stream** intermediate representation to the polyhedral representation used by the mapper. Finally, in Sections 19, 20 and 21 we describe the details of code generation for specific architectures.

2 The Polyhedral Model

The polyhedral model is a linear-algebraic mathematical formalism used in the R-Stream mapper for modeling and transforming *static control programs*. Static control programs are composed of do-loops with loop lowerbounds, loop upperbounds, and array indexing functions that are affine functions of outer loop indices and unknown symbolic parameters.

We can summarize the class of programs using the following Backus-Naur Form:

```

parameters ::= M, N, ...
arrays      ::= A, B, C, ...
loop indices ::= i, j, k, ...
e           ::= affine expressions of indices and parameters
S           ::= A[e] = f(A1[e1], ..., An[en])
              | for i = e ... e do S
              | S; S; ...

```

2.1 Polyhedra and polytopes

As evident from its name, in the polyhedral model we are mainly interested in modeling our loop nests using polyhedra and objects related to polyhedra. A *polyhedron* can be specified in various equivalent ways. For example, we can define a polyhedron as the conjunction of m linear constraints $\{x \in \mathbb{R}^n \mid Ax + b \geq 0\}$ where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$, or explicitly as a Minkowski sum of *lines* $\{l_1, \dots, l_n\}, n \geq 0$, *rays* $\{r_1, \dots, r_m\}, m \geq 0$, and *vertices* $\{v_1, \dots, v_k\}, k \geq 1$:

$$P = \text{lin.space}\{l_1, \dots, l_n\} + \text{cone}\{r_1, \dots, r_m\} + \text{convex}\{v_1, \dots, v_k\} \quad (1)$$

where

$$\text{lin.space}\{l_1, \dots, l_n\} = \{\sum_{1 \leq i \leq n} \alpha_i l_i \mid \alpha_i \in \mathbb{R}, i \leq 1 \leq n\} \quad (2)$$

$$\text{cone}\{r_1, \dots, r_m\} = \{\sum_{1 \leq i \leq m} \beta_i r_i \mid \beta_i \geq 0, 1 \leq i \leq m\} \quad (3)$$

$$\text{convex}\{v_1, \dots, v_k\} = \{\sum_{1 \leq i \leq k} \gamma_i v_i \mid 0 \leq \gamma_i \leq 1, \sum_{1 \leq i \leq k} \gamma_i = 1\} \quad (4)$$

A polyhedron lacking lines and rays is bounded, and is called a *polytope*.

2.1.1 Parameters

The polyhedral model can elegantly model programs with symbolic integer parameters using *parametric polyhedra*, as long as these parameters are only used in an affine context. For example, a parametric polyhedron with p parameters can be defined in the similar manner as a polyhedron, as the intersection of a suitable number of constraints: $\{x \in \mathbb{R}^n \mid Ax + By + b \geq 0\}$ where $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{m \times p}$ and $b \in \mathbb{R}^m$.

2.1.2 Z-polyhedra

Since loop indices and array indices are integers, we frequently have to restrict ourselves to consider only integral points within a (parametric) polyhedron. A *Z-polyhedron* Z is the intersection of a polyhedron P with an integer lattice L , where L can be defined as $\{Ax + b \geq 0 \mid x \in \mathbb{Z}^d\}$ for some appropriate matrix A and vector b .

2.1.3 Domain and Z-domain

A polyhedron is a convex set. However, many of the sets that appear in mapping, such as data footprints and communication sets, are non-convex. Thus for convenience, we frequently manipulate domains and Z-domains rather than polyhedra and Z-polyhedra. A *domain* is a finite union of polyhedra, while a *Z-domain* [QRR96, NR00] is a finite union of Z-polyhedra. Both domain and Z-domain are closed under set operations such as union, intersection, difference, and image and preimage operations under affine functions [GR07, SL06, SLM07]. In terms of the same expressiveness, Z-domains are identical to Presburger sets.

2.2 Modeling iteration spaces

There are a few general steps when using all the above machinery to model a static control program.

The first step is to assign each iteration of a statement a unique coordinate. We can accomplish this as follows. Suppose we are given a do-loop program with statements S_1, \dots, S_n .

An iteration of a statement is called an *operation*, which can be specified by annotating the statement with the values of all the loop counters in which the statement is nested, from the outermost to the innermost. We can write this as $\langle S, x \rangle$. The *iteration vector* x represents the value of the loop counters. Its number of dimensions is the number of loop counters.

The iteration vector of a statement S is constrained by all the loop bounds which enclose it (i.e., its *iteration space* or *iteration domain*.)

For example, the loop nests below contains two statements, S1 and S2.

```
for (int i = 0; i < n; i++) {
  for (int j = i+1; j < m; j += 2) {
    A[i][j] = i+j; // S1
  }
  for (int j = i+1; j < min(m, i+10); j++) {
    A[i][j] += 10; // S2
  }
}
```

The iteration spaces of the two statements are:

$$\mathcal{D}_1 = \{[i, j] \in \mathbb{Z}^2 \mid \exists k \in \mathbb{Z} \cdot \mathbb{Z}^2 \mid 0 \leq i < n, i+1 \leq j < m, j-i-1 = 2k\} \quad (5)$$

$$\mathcal{D}_2 = \{[i, j] \in \mathbb{Z}^2 \mid i \leq 0 < n, i+1 \leq j, j < m, j < i+10\} \quad (6)$$

Note that both \mathcal{D}_1 and \mathcal{D}_2 are parametric in terms of the symbolic unknowns m and n .

2.3 Modeling dependences

The second step to model a static control program is to identify when two operations may be executed in parallel, or when one must precede another because of a producer-consumer relationship. We do this by capturing all the dependences. Given any two statements S and T , let \mathcal{R}_{ST} denote the dependence relation between operations in S and T . That is, if $\langle S, x \rangle$ depends on $\langle T, y \rangle$, then $(x, y) \in \mathcal{R}_{ST}$.

We can define \mathcal{R}_{ST} in a static control program precisely as follows. Let the *sequencing predicate* $\langle S, x \rangle \prec \langle T, y \rangle$ denotes true iff operation $\langle S, x \rangle$ is executed before $\langle T, y \rangle$ in the original loop nests. Suppose statement S contains only one reference to $A[f_S(x)]$ and statement T contains only one reference to $A[f_T(x)]$, and the two references induce a dependence.⁴ Using the relation \prec and the iteration spaces of S and T , we can define \mathcal{R}_{ST} as:

$$\mathcal{R}_{ST} = \left\{ [x, y] \mid \begin{array}{l} f_S(x) = f_T(y), \\ x \in \mathcal{D}_S, \\ y \in \mathcal{D}_T, \\ \langle S, x \rangle \prec \langle T, y \rangle \end{array} \right\} \quad (7)$$

The same approach can be generalized if S and T contains multiple references; \mathcal{R}_{ST} simply contains the union of all the dependence relations computed from each pair of references.

The sequencing predicate \prec can be defined from the structure of the loop nests as follows. Given S and T , let N_{ST} be the maximal depth of the loop nest in which S and the T . Let T_{ST} be a boolean flag which is true iff S precedes T textually in the program. Then we can define \prec as:

$$\begin{aligned} \langle S, x \rangle \prec \langle T, y \rangle &\equiv x[1 \dots N_{ST}] \prec_{lex} y[1 \dots N_{ST}] \vee \\ &\quad (x[1 \dots N_{ST}] = y[1 \dots N_{ST}] \wedge T_{ST}) \end{aligned} \quad (8)$$

where \prec_{lex} is the lexicographical order.

Note that when the program input falls under scope of static control programs, dependence relations represent a program's dependences exactly. Other dependence representations in traditional parallelization compilers, such as dependence vectors and direction vectors, are merely approximations.

⁴That is, at least one reference is a write reference.

2.4 Space-time mappings

Finally, we would like to specify the temporal order for all the operations that the processor will perform and where they should be performed. In fact, mapping, in the simplest sense, is nothing more than specifying these two attributes.

One of the most general ways to do this is to associate each operation $\langle S, x \rangle$ with a *space-time mapping*:

$$\Theta_S(x) = \begin{bmatrix} s(x) \\ t(x) \end{bmatrix}$$

where $s(x)$ specifies the processor element where operation $\langle S, x \rangle$ is to be executed, and $t(x)$ specifies its execution time. Note that we allow both s and t to be multidimensional functions. Here is how s and t can be interpreted.

For the former $s(x)$, we can interpret the co-domain of the function as the processor space. For example, if the target of our mapping is a two dimensional processor grid of size $N \times M$, then the most natural co-domain of s is a set of $\mathbb{Z}_N \times \mathbb{Z}_M$.

For the latter $t(x)$, we can interpret “time” as a multiple dimensional space. Note that time in this sense is often merely a logical device for imposing order; that is, it does not have to correspond to any physical time units. The only imposition we make of it is that the logical time obeys a total *lexicographical* order, that is, given $t(x) \prec_{lex} t(y)$, then operation $\langle S, x \rangle$ executes before $\langle S, y \rangle$ in the mapping.

For a space-time mapping to be correct, it must preserve the dependences in the original program, i.e.,

$$\forall (x, y) \in \mathcal{R}_{ST}, \Theta_S(x) \succ_{lex} \Theta_T(y) \quad \text{for all statements } S \text{ and } T. \quad (9)$$

where \mathcal{R}_{ST} stands for the appropriate dependence relation between statement S and T . This is essentially the mathematically equivalent of the statement: “if operation $\langle S, x \rangle$ depends on operation $\langle T, y \rangle$, $\langle S, x \rangle$ must be scheduled after $\langle T, y \rangle$.” That is, dependences must be preserved.

2.5 Example

As an example, consider the following simple loop nests:

```
for (i = 1; i <= n; i++) {
  for (j = 1; j <= n; j++) {
    a[i][j] = a[i][j-1] + a[j][i]; // S
  }
}
```

The iteration domain of the statement S is $\mathcal{D} = \{[i, j] \in \mathbb{Z}^2 \mid 1 \leq i \leq n, 1 \leq j \leq n\}$.

Since there is only one statement, there is only one dependence relation \mathcal{R} , computed from two pairs of references ($\mathbf{a}[\mathbf{i}][\mathbf{j}]$, $\mathbf{a}[\mathbf{i}][\mathbf{j}-1]$), and ($\mathbf{a}[\mathbf{i}][\mathbf{j}]$, $\mathbf{a}[\mathbf{j}][\mathbf{i}]$).

$$\mathcal{R} = \{[[i, j], [i', j']] \mid i = i', j = j' - 1, [i, j] \in \mathcal{D}, [i', j'] \in \mathcal{D}, \langle S, [i, j] \rangle \prec \langle S, [i', j'] \rangle\} \cup \{[[i, j], [i', j']] \mid i = j', i = j', [i, j] \in \mathcal{D}, [i', j'] \in \mathcal{D}, \langle S, [i, j] \rangle \prec \langle S, [i', j'] \rangle\}$$

According to the definition, the relation \prec can be written as:

$$\langle S, x \rangle \prec \langle S, y \rangle \equiv x \prec_{lex} y$$

The above can be simplified as rewritten as the union of the following two relations:

Target	Source	Dependence Relation	Type
$\mathbf{a}[\mathbf{i}, \mathbf{j}]$	$\mathbf{a}[\mathbf{j}, \mathbf{i}]$	$\{([i, j], [j, i]) \mid 1 \leq j, i \leq n, -j + i - 1 \geq 0\}$	true, anti
$\mathbf{a}[\mathbf{i}, \mathbf{j}]$	$\mathbf{a}[\mathbf{i}, \mathbf{j}-1]$	$\{([i, j+1], [i, j]) \mid 1 \leq j \leq n-1, 0 \leq i \leq n\}$	true

We can obtain the following 1-D scheduling which satisfies the above dependences:

$$\Theta(i, j) = 2i + j$$

It can be interpreted simply: iteration (i, j) of the loop nests is to be executed at time step $2i + j$. For example, at step 4 iterations $(0, 4)$, $(1, 2)$, and $(2, 0)$ should be executed.

This schedule, when rendered in traditional loop nests form, actually represents the following parallelized program fragment:

```
for (i = 3; i <= 3 * n; i++) {
  doall (j = max(i-n, ⌊(i+1)/2⌋);
        j <= min(i-1, ⌊(i+n)/2⌋); j++) {
    a[i-j][2*j-i] = a[i-j][2*j-i-1] + a[2*j-i][i-j];
  }
}
```

In this simple example, we can show that the schedule is valid according to (9) by verifying that these constraints hold manually:

$$\Theta(i, j) - \Theta(j, i) > 0 \quad \text{for all } 1 \leq j, i \leq n, -j + i - 1 \geq 0 \quad (10)$$

$$\Theta(i, j+1) - \Theta(i, j) > 0 \quad \text{for all } 1 \leq j \leq n-1, 0 \leq i \leq n \quad (11)$$

Constraint (10) is equivalent to:

$$\begin{aligned} (2i + j) - (2j + i) &> 0 \quad \text{for all } 1 \leq j, i \leq n, -j + i - 1 \geq 0 \\ i - j &> 0 \quad \text{for all } 1 \leq j, i \leq n, -j + i - 1 \geq 0 \end{aligned}$$

and constraint (11) simplifies to:

$$\begin{aligned} (2i + j + 1) - (2i + j) &> 0 \quad \text{for all } 1 \leq j \leq n - 1, 0 \leq i \leq n \\ 1 &> 0 \quad \text{for all } 1 \leq j \leq n - 1, 0 \leq i \leq n \end{aligned}$$

How Θ can be computed automatically will be described in Section 7, where we present the scheduling algorithms in the **R-Stream** mapper.

2.6 Further Readings

A full introduction to the polyhedral model is beyond the scope of this report. Interested readers may refer to [Fea96], [Len93], [DRV00], and [Vas07](Chapter 2) for more comprehensive introductions to this rich topic.

3 Mapper Architecture

Figure 1 shows the basic structure of the current mapper in relation to the rest of the R-Stream compiler infrastructure. To make it possible to develop the mapper and the compiler as two separate entities, the mapper is structured as a self-contained (albeit highly complex) optimization phase, from the view of the R-Stream compiler. The core compiler infrastructure and the mapper communicate with each other only through a narrow interface.

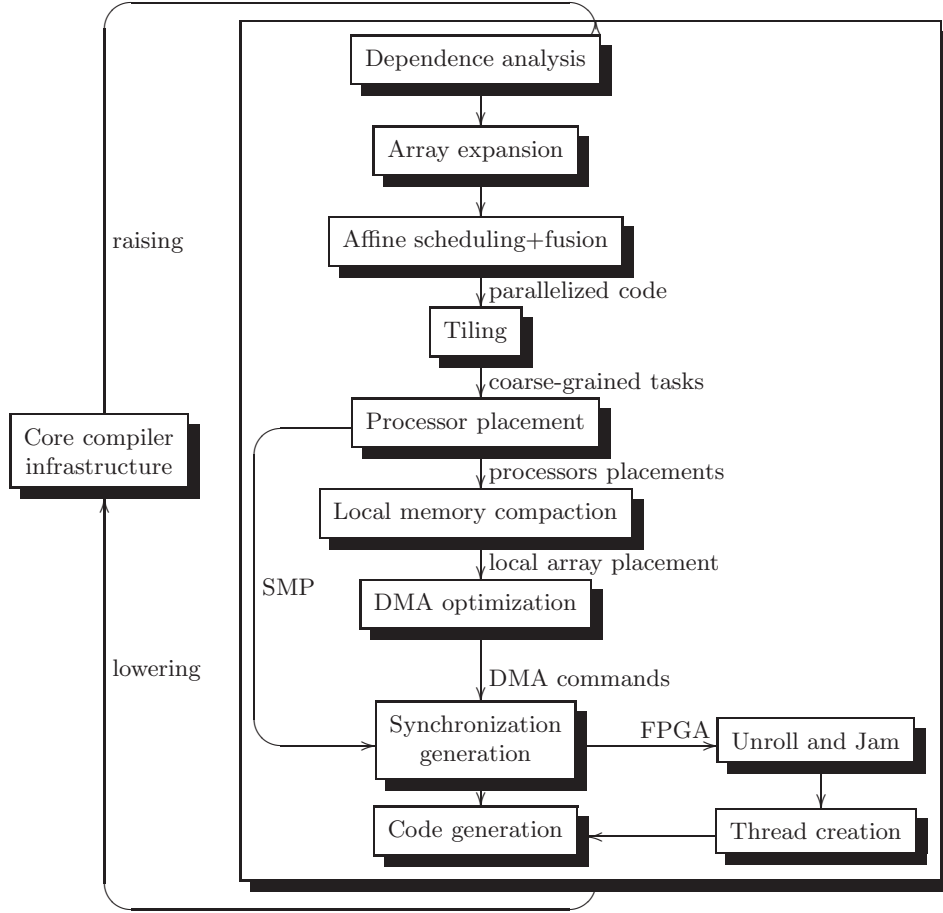


Figure 1: Mapper Architecture.

The core compiler infrastructure is responsible for providing the following important services:

Scalar optimizations Classical SSA-based scalar optimizations, such as conditional constant propagation [WZ91], value numbering [Cli95, CS95],

strength reduction [CSV01], etc. are performed on the input programs in order to eliminate redundancies in the input code.

Region selection The compiler has to determine which regions of code in the application program can and should be processed using the mapper. We call such sections of code *mapped regions*.

Raising Mapped regions of code are translated into the polyhedral form required by the mapper.

Lowering After mapping, the mapped code in polyhedral form are translated back into loop nests required by the compiler through the code generation phase. Special target dependent APIs may be inserted at this point. If required, the compiler can also perform traditional optimizations to reduce redundancies in the mapped code as a post-processing step.

Idiomatic target code output Aside from aesthetic considerations, we have found that the binary produced by many LLCs (Low-level Compilers, the compilers that accept the output of R-Stream) performs significantly worse when taking complex machine-generated C code as input than when compiling code written by a person. Those LLCs “key” on specific syntactic patterns for idiomatic transformation strategies that are lost when semantically equivalent code does not use those patterns. Thus a high level syntax recovery [Cif93, Cif94] is performed to produce output code that resembles idiomatic C written by a human rather than the internal dump of a compiler.

If we delve into the structure of the mapper, the following subcomponents appear:

Dependence analysis This phase analyzes the program in question and generates the appropriate dependence graph.

Array expansion The array expansion phase is used to improve the available parallelization by performing *scalar and array expansion*.

Affine scheduling and fusion The next step is affine scheduling. It extracts coarse grained parallelism from the input loop nests. Next, a fusion step is performed, which attempts to improve the locality of reference of the parallelized code.

Tiling The affine scheduling phase generates coarse-grained but resource-unconstrained parallelism. In order to ensure that the data footprint of the loop nests fit into the available memory of the target processor, the iteration spaces of the statements are partitioned into hyperrectangular “tiles” in the tiling phase. The tiling process attempts to choose the proper size and shape so as to maximize the locality of reference and utilization of memory while minimizing communication. Each of the resulting tiles form a logical task to be executed on a process as a single unit.

Processor placement The tiling phase assumes an infinite number of processors in a virtual processor space, while real hardware, of course, contains a finite number of these. A typical architecture is structured as a regular grid in \mathbb{Z}^p , where p is typically 1 or 2. In the processor placement phase, we fold the virtual processor space in \mathbb{Z}^d , where d is the number of dimensions in the loop nests, back into a finite processor grid in \mathbb{Z}^p .

Distributed memory mapping On a distributed memory architecture, each processor has an associated local memory. The mapper has to perform extra local data layout and communication generation tasks. These are:

Local memory compaction This phase takes the processed loop nests and attempts to migrate all array references to local memory. The physical layout of the (local) arrays and loop access expressions can be altered in this process.

Communication and DMA generation When migrating array references from global to local memory, we also have to insert communication operations at the appropriate places. To ensure that all communication is coarse-grained in nature, we batch all communications and only insert such communications at tile boundaries. To further enhance the effectiveness of the mapper, we use multi-buffering to prefetch data, and maximize the overlap of computation and communication. Communication operations are subsequently mapped into the target architecture-provided DMA operations. This mapping step chooses a suitable set of DMA operations, constrained by the capabilities of the hardware.

These tasks are omitted when mapping for a symmetric multiprocessor (SMP) architecture.

Synchronization generation The synchronization phase inserts synchronization primitives to ensure proper execution of the mapped program.

Jam A form of unroll and jam (without the unrolling) is used to extract extra fine-grained vector and pipeline parallelism. These are useful for SIMD and FPGA targets.

FPGA mapping For mapping certain FPGA-accelerated targets, that have an execution model where the accelerator is tightly coupled to the host, an extra component is needed:

Thread generation in the FPGA targets we have considered, FPGA requires DMA and synchronization to be performed on the host processor. Thread generation is the process of splitting the parallelized loop nest into two parts: (i) computation-only kernels to be mapped onto the FPGA, (ii) synchronization and DMA operations to be placed on the host processor.

Code generation Finally, in the code generation phase we convert the mapped program in polyhedral form back into loop nests.

4 The Generalized Dependence Graph

The main intermediate representation of the R-Stream mapper is the *generalized dependence graph (GDG)*. All phases of the mapper take the GDG as an input and produce a modified GDG as output. Unlike more traditional and syntactically oriented representations for loop level optimizers, such as abstract syntax trees, the GDG summarizes all the relevant details of a set of loop nests to be optimized in a mathematical form suitable for polyhedral optimizations.

A GDG is a multi-graph $G = (V, E, C)$ where the nodes V are the set of statements in the program to be mapped, and $E \in V \times V$ are the set of dependence edges between these statements, and $C \subset \mathbb{Z}^{gp}$ is the *context* of the system, and gp is the number of *global parameters*⁵ where the context represents constraints on the global parameters or system parameters.

To each statement $S \in V$ in the GDG, we attach the quadruple $(\mathcal{D}_S, R_S, p_S, \Theta_S)$, where \mathcal{D}_S is the iteration domain of S , R_S is a set of array references of the form $A[g(x)]$, p_S is a predicate function on the subset of array references R_S , and $\Theta : \mathbb{Z}^{d+gp} \rightarrow \mathbb{Z}^{2d+1}$ is the space-time mapping of S . The purposes of these components are:

- The iteration domain is used to encode the loop nests under which S is nested.
- The set of array references R_S encodes all the array and scalar references within the statement.
- The predicate function p_S encodes statements which have been if-converted [AKPW83], i.e., having its control dependences converted into data dependences.
- Finally, the space-time mapping is a function that determines at which processor and at which time a statement instance should be executed. Initially, the space component of the function is a constant and the time component is the identity function (which means the program will be executed sequentially on one processor as in the input). As the mapping process progresses, it will be refined as more parallelism is exposed.

4.1 Space-time mapping

The space-time mapping Θ contains the following restrictions: every odd dimension is a constant function. Θ can be further decomposed into three components (α, β, γ) , following the convention described in [Kel96, KPR98, KPR95,

⁵Global parameters are also referred to as system parameters or structural parameters in the literature.

BCG⁺03a, BCG⁺03b]. In matrix form this is the following:

$$\Theta(x, y) = \begin{bmatrix} 0 & \dots & 0 & 0 & \dots & \beta_1 \\ \alpha_{1,1} & \dots & \alpha_{1,d} & \gamma_{1,1} & \dots & \gamma_{1,gp+1} \\ 0 & \dots & 0 & 0 & \dots & \beta_2 \\ \alpha_{2,1} & \dots & \alpha_{2,d} & \gamma_{2,1} & \dots & \gamma_{2,gp+1} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & 0 & \dots & \beta_d \\ \alpha_{d,1} & \dots & \alpha_{d,d} & \gamma_{d,1} & \dots & \gamma_{d,gp+1} \\ 0 & \dots & 0 & 0 & \dots & \beta_{d+1} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_d \\ y_1 \\ \vdots \\ y_{gp} \\ 1 \end{bmatrix}$$

The interpretation of α , β and γ are as follows. The α component describes a linear transformation from $\mathbb{Z}^d \rightarrow \mathbb{Z}^d$ on x , i.e., on the loop indices. The β -vector component represents $d+1$ constant functions, and can be interpreted as the shape of an (imperfectly) nested loop nest. Finally, the γ component is an affine transformation from $\mathbb{Z}^{gp} \rightarrow \mathbb{Z}^d$, and these represent the amount of loop shifting in the transformation. We call the odd dimensions of Θ β -dimensions and the rest non- β dimensions.

Note that this representation contains redundant information; it is possible to represent a similar transformation with only d dimensions. However, this representation is more convenient for program transformations. For example, fusing or fissioning a loop can be performed by modifying only the β component of a schedule. Similarly, loop shifting or alignment only modifies the γ component, and traditional unimodular loop transformations (skewing, interchange, and reversal) only affect the α component.

To represent parallelism and processor assignment, we extend the space/time mapping representation with the following attributes:

parallelism kind vector Parallelism in the GDG is indicated by attaching each non- β dimension of Θ with a parallelism kind indicator. Currently, the kinds of parallelism used in our implementation are only “sequential,” and “doall.” It is possible, however, to represent pipeline parallelism within the representation for future versions of the mapper that will support it.

task group id To deal with the possibility of representing groups of tasks with atomic computation phases, each statement $S \in V$ is assigned a group id $g_S : \mathbb{Z}$ with the interpretation that all statements with the same group id belong to the same abstract task. Task group ids are assigned by the tiling phase in the mapper.

processor dimensions A processor placement algorithm may associate certain loop dimensions of a statement to correspond to the processors of the target processor grid. A dimension range $[proc_begin, proc_end)$ may be attached to each statement to indicate such information.

Similar extra processor dimensions may be attached to the access functions of individual arrays within the mapped program to indicate a distributed array.

heterogeneous system id Heterogeneous systems may contain multiple processor grids, each belonging to different system. To accomodate this we also allow a grid id to be attached to each statement. This attribute indicates which processor grid within a heterogeneous system a statement is placed in mapped form.

4.2 Dependence edges

Each edge $(S, T) = e \in E$ in the GDG represents the dependence between the two statements S and T . Attached to each edge is a *dependence Z-polyhedron* \mathcal{R}_e , which summarizes the dependences between operations of S and T , i.e.,

$$\mathcal{R}_e = \{(i, j) \mid \langle S, i \rangle \text{ depends on } \langle T, j \rangle\}$$

4.3 Example 1

The ideas in the previous section can be clarified with a concrete example. Consider the following one statement loop nest:

```
for (i = 1; i <= n; i++) {
  for (j = 1; j <= n; j++) {
    a[i][j] = a[i][-1 + j] + a[j][i]; // S
  }
}
```

This set of loop nests contains only one global parameter n . In this example, we can assume the value of n is greater than or equal to 1. Thus the context of the GDG \mathcal{C} is the polyhedron $\{n \mid n \geq 1\}$. The GDG $G = (V, E, \mathcal{C})$ contains only one node $V = \{S\}$, where S represents the statement of the same name within the loop. The iteration space of S is $\mathcal{D}_S(n) = \{[i, j] \in \mathbb{Z}^2 \mid 1 \leq i \leq n, 1 \leq j \leq n\}$. By performing dependence analysis using the process outlined in the previous section, we can determine that the loop has the following set of dependencies.

Target	Source	Dependence Relation	Type
$a[i, j]$	$a[j, i]$	$\{([i, j], [j, i]) \mid 1 \leq j, i \leq n, -j + i - 1 \geq 0\}$	true, anti
$a[i, j]$	$a[i, j-1]$	$\{([i, j+1], [i, j]) \mid 1 \leq j \leq n-1, 0 \leq i \leq n\}$	true

The initial schedule for the statement within the loop is the “identity” function:

$$\Theta_S(i, j, n) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ n \\ 1 \end{bmatrix}$$

Θ_S can be decomposed into its α , and β and γ constituents:

$$\begin{aligned}\alpha_S &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \beta_S &= [0, 0, 0]^T \\ \gamma_S &= \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}\end{aligned}$$

4.4 Example 2

Below we consider another loop nests example with two statements.

```
for (int i = 1; i <= T; i++) {
  for (int j = 2; j <= -1 + N; j++) {
    for (int k = 2; k <= -1 + N; k++) {
      L[j][k] = f(A[j][k+1], A[j][k-1], A[j+1][k], A[j-1][k]); // 1
    }
  }
  for (int j = 2; j <= -1 + N; j++) {
    for (int k = 2; k <= -1 + N; k++) {
      A[j][k] = L[j][k]; // 2
    }
  }
}
```

The variables T and N are symbolic unknowns to the loops and are thus represented as global parameters within the GDG. The iteration spaces of the two statements are identical, and are given below.

$$\begin{aligned}\mathcal{D}_1 &= \{[i, j, k] \mid 1 \leq i \leq T, 2 \leq j \leq N-1, 2 \leq k \leq N-1\} \\ \mathcal{D}_2 &= \{[i, j, k] \mid 1 \leq i \leq T, 2 \leq j \leq N-1, 2 \leq k \leq N-1\}\end{aligned}$$

Due to space limitations, we shall omit listing all the dependence relations.

The initial schedules for the two statements are:

$$\begin{aligned}
\Theta_1(i, j, k, T, N) &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \\ T \\ N \\ 1 \end{bmatrix} \\
\alpha_1 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
\beta_1 &= [0, 0, 0, 0]^T \\
\gamma_1 &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\
\Theta_2(i, j, k, T, N) &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \\ T \\ N \\ 1 \end{bmatrix} \\
\alpha_2 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
\beta_2 &= [0, 1, 0, 0]^T \\
\gamma_2 &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\end{aligned}$$

Note that the two schedules are not identical. The third dimension of Θ_1 is 0 while in Θ_2 it is 1. This reflects the fact that the two statements are nested under the first loop nest i but not under the second j loop nests.

One of the classic optimizations that we can do is performing loop fusion to improve the spatial locality of the loop nests. In the above example, we can do this by merging the two statements into a common set of loops. However, due to a potential dependence violation, we have to “shift” the first statement ahead by one i -iteration in order to preserve dependencies. The resulting loop nests contains a prologue and an epilogue resulting from this transformation:

```

if (N >= 3)
  for (int i = 1; i <= T; i++) {
    for (int j = 2; j <= N + -1; j++) {
      L[2][j] = f(A[2][j+1], A[2][j-1], A[3][j], A[1][j]);

      for (int j = 2; j <= N + -2; j++) {

```

```

    for (int k = 2; k <= N + -1; k++) {
        L[1+j][k] = f(A[j+1][k+1], A[j+1][k-1], A[j+2][k], A[j][k]);
        A[j][k] = L[j][k];
    }
}
for (int j = 2; j <= N + -1; j++) {
    A[N-1][j] = L[N-1][j];
}
}

```

While the loop nests in syntactic form requires somewhat non-trivial generation of prologue and epilogue, it is trivial to represent the result of this transformation using the internal space-time mapping notation:

$$\begin{aligned}
 \Theta_1(i, j, k, T, N) &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \\ T \\ N \\ 1 \end{bmatrix} \\
 \alpha_1 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
 \beta_1 &= [0, 0, 0, 0]^T \\
 \gamma_1 &= \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\
 \Theta_2(i, j, k, T, N) &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \\ T \\ N \\ 1 \end{bmatrix} \\
 \alpha_1 &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
 \beta_1 &= [0, 0, 0, 1]^T \\
 \gamma_1 &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
 \end{aligned}$$

We can interpret these matrices as follows. The second dimension of Θ_1 is changed from i to $i-1$, while that of Θ_2 remains at i . This reflects the fact that

we want to execute the statement 1 one i iteration ahead than before. We also transform the third dimension of Θ_2 from 1 to 0, which denotes the fact that we want to fuse the two statements under loop nest j (and k). Finally, the last dimension of Θ_2 is changed from 0 to 1, which denotes the fact that statement 2 should be executed after statement 1 within the i, j, k loop nests.

4.5 Related Works

The notion of a generalized dependence graph is taken from various works of Feautrier [Fea91, Fea92a, Fea92b]. The representation of a schedule as its α , β and γ components is taken directly from the WrapIt system [BCG⁺03a], although the use of extra constant scheduling dimensions (β) to encode the loop nesting structure was first used extensively in the work Kelly et al. [Kel96, KPR98, KPR95].

5 Polyhedral Mapper Infrastructure

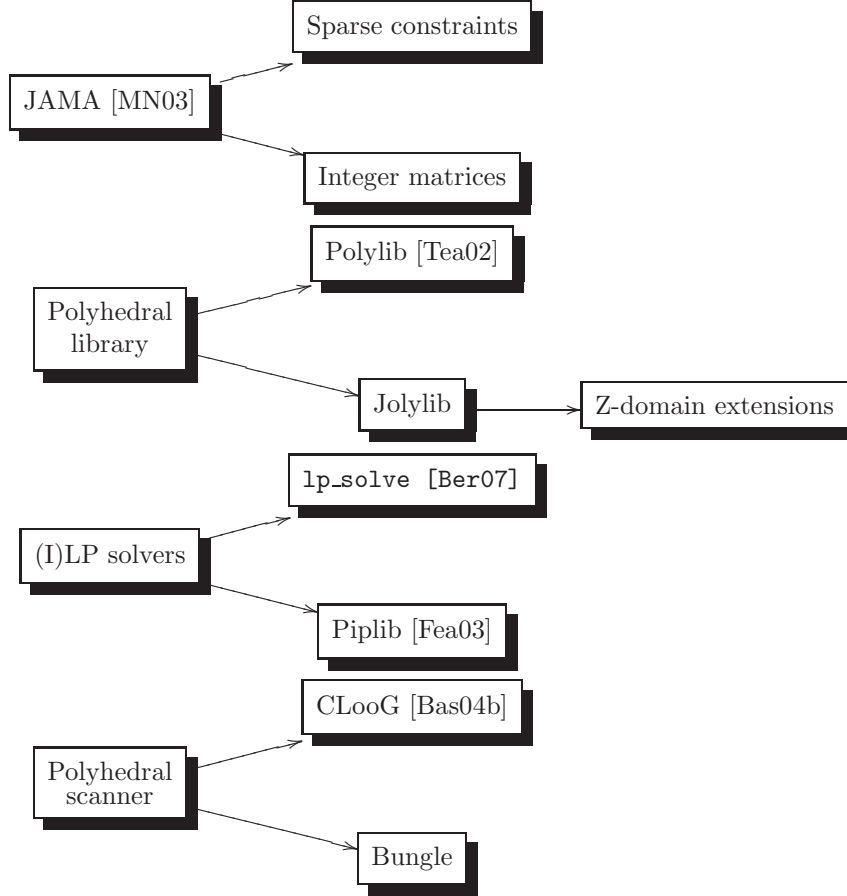


Figure 2: Polyhedral mapper software infrastructure.

The R-Stream mapper requires only a few software libraries and tools developed externally outside of Reservoir Labs. A number of libraries are available in the field of high level optimization. However, most of them did not meet our expectations in terms of robustness, scalability or software engineering. Hence, we have developed our own versions of these libraries, which use better or equivalent algorithms and are more robust. We wrote them in Java for homogeneousness of our software architecture and productivity.

Figure 2 shows how we organize these libraries and tools within the mapper. External libraries written in C and C++ are interfaced with Java via the Java Native Interface (JNI), and then encapsulated as Java classes within the R-Stream mapper environment.

Some of the libraries shown are released under the GPL. We have the option to use them internally for development purposes. For production and distribution, we use proprietary libraries that have greater performance, higher stability, and specific feature sets needed by the polyhedral mapper.

The libraries and tools can be categorized as follows:

Matrix library The R-Stream mapper makes heavy use of matrices and constraints encoded as matrices. We start with **JAMA**, a public domain matrix library distributed by MathWorks and National Institute of Standards and Technology (NIST). Since this core library only contains double floating point matrices and related algorithms, we have substantially extended the library to include integer matrices and operations on integer matrices.⁶

Polyhedral library Many operations in the R-Stream mapper are manipulated as polyhedral sets.

- We started with **Polylib** [Wil93, Tea02], a polyhedral library providing convenient operations on polyhedral sets such as union, difference, intersection and projections. The GNU multiprecision numeric library (GMP) is used internally in Polylib for manipulating big integers.
- While Polylib has proven to be very useful, the fact that it is written in C has proven to be an obstacle in developing extensions. We have redesigned **Polylib** adding critical performance optimizations and provide our own efficient standalone library in Java called **Jolylib**. Recently, we added a Jolylib extension to Z-domains using the algorithms in [Pug92, SL06, SLM07, GR07]. Z-domain extensions are based on previous seminal contributions [QRR96, NR00].

Integer linear programming solvers The R-Stream mapper makes use of linear programming and integer linear programming in scheduling, local memory compaction, array compaction, and dataflow analysis. For these tasks, the following tools are used:

- **Lpsolve**, an open source linear programming and integer linear programming solver.
- **Piplib** [Fea88b, Fea03], a parametric integer linear programming tool developed by Feautrier et al. Because **Piplib** calls are very expensive, and due to GPL constraints, we only link it in and use it in for experimentation. We are considering replacing this with a Java version, with more modern algorithms.
- We are considering moving to using **CPLEX**, a commercial mixed integer linear programming (MILP) tool.

⁶These include computation of Hermite Normal Form and Smith Normal Form, lattice and subspace operations, and lattice basis reduction algorithms.

Genetic Programming The tiling component (Section 8) of the R-Stream mapper uses genetic programming to select the optimal tile sizes. We are currently using the **JGap** framework [dt07] for this purpose.

Polyhedral scanner Polyhedral scanning is the process of synthesizing loop nests from the internal polyhedral representation (Section 15). Two tools are used:

- **CLooG** [Bas04b], a polyhedral scanner developed by Cedric Bastoul based on the Quilleré separation algorithm [QRW00],
- **Bungle**⁷, our implementation of the Quilleré algorithm with critical emphasis put on code quality improvements and code generation speed.

⁷Bungle is a pun on the pronunciation of **CLooG** as “kludge.”

6 Array Expansion

Array expansion is the process of elimination of false dependences (i.e., output- and anti-dependences) by expanding the size and/or dimensions of arrays so that conflicting writes in different loop iterations are given distinct array locations. Such transformations are also a precursor to parallelization enhancements such as array privatization [GLL95, Tu95, MAL93, TP01], where arrays holding temporary values are replicated on each processor.

Array expansion is desirable, because programs written for a single-thread environment are often not directly suitable for parallelization. For example, consider the following manually “optimized” matrix multiply loop kernel, where the programmer uses a single scalar variable `s` to hold the running value of the dot product:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        s = 0;
        for (k = 0; k < N; k++) {
            s += A[i][k] * B[k][j];
        }
        C[i][j] = s;
    }
}
```

The intention, of course, is that the scalar variable can be mapped directly into a machine register, reducing the number of memory accesses. However, if parallelization is the goal, the single scalar `s` becomes a dependence bottleneck, since all iterations of the loop nests have to read and write to `s`.

One possible transformation that can eliminate the false dependences on `s` is scalar expansion, a degenerate case of array expansion restricted to scalars. For example, by expanding the scalar `s` into two dimensional array, we can obtain the following loop nests:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        s[i][j] = 0;
        for (k = 0; k < N; k++) {
            s[i][j] += A[i][k] * B[k][j];
        }
        C[i][j] = s[i][j];
    }
}
```

In the above loop nests, all iterations of loops `i` and `j` can be executed in parallel because each (i, j) -iteration read and writes to different locations of `s`. The array

s above can actually be replaced completely by `C`. This can be accomplished by *array copy propagation*, a secondary transformation, after array expansion.

Scalar expansion can be generalized to arrays. The following example, taken from [Fea88a], demonstrates this transformation. We are given a sequence of loop nests operating on an array `c`.

```
for (k = 0; k <= m+n; k++) {
  c[k] = 0;
  for (i = 0; i <= m; i++) {
    for (j = 0; j <= n; j++) {
      c[i+j] = c[i+j] + a[i] * b[j];
    }
  }
}
```

The false dependences on `c` can be removed by replacing `c` in the second loop nest with the 2-dimensional array `cc`. The remaining loop nests contain only true dependences.

```
for (k = 0; k <= m+n; k++) {
  c[k] = 0;
  for (i = 0; i <= m; i++) {
    for (j = 0; j <= n; j++) {
      if (n-j-1 < 0 || i-1 < 0) {
        cc[i,j] = c[i+j];
      } else {
        cc[i,j] = cc[i-1,j+1] + a[i] * b[j];
      }
    }
  }
}
```

6.1 Related works

Feautrier [Fea88a] describes the first algorithm for array expansion in the polyhedral model using a variant of parametric integer programming [Fea88b]. Recent improvements to the Feautrier algorithm includes [BCC98]. One of the crucial problems to array expansion is the array data dependence analysis problem [Fea91, MAL93, CBF95]. Closely related to the problem of array expansion is *array privatization*, which are described in various works in the literature, e.g., [GLL95, Tu95, MAL93, TP01].

6.2 Current algorithm

Currently, we implement an interim version of array expansion, expanding only a subset of the potential values. This is due to project considerations; it allowed us to build a version of array expansion prior to our engineering of array dataflow

analysis [Fea91, MAL93, CBF95]. Our interim array expansion is described below; with the recent robust support for Z-Domains, we will be able to build the needed array dataflow analysis.

An *expandable temporary* is a variable whose value is assigned in the same iteration as its use. This is a very natural by-product of sequential programs. Unfortunately, it systematically creates false dependences between iterations.

This is defined by the following conditions.

1. All references to the expanded temporary must be nested under L common loop nests. This means that the β -strings of all references have a common prefix of at least length L .
2. All locations read by a reference r must be written by some other references before r is executed within the current iteration.

To be more precise, let y denote the set of system parameters. Let the read references be $(f_1, D_1), \dots, (f_n, D_n)$. Let the write references be $(g_1, D'_1), \dots, (g_m, D'_m)$. Given an iteration vector i , let i_L denote the length L prefix of i . Let $(S, i) \prec (T, j)$ denote the fact that iteration i of statement S precedes iteration j of T .

Suppose we have one write reference (g, D'_w) (in statement S) and one read reference (f, D_r) (in statement T). Then consider

$$R = \{(a, b, i, j) | a = f(i, y), b = g(j, y), i \in D'_w(y), j \in D_r(y), (S, i) \prec (T, j), i_L = j_L\} \quad (12)$$

This set represents the set of all locations written by the statement S before being read by the statement T within loop nesting depth L .

To enforce the fact that all locations read in the iteration must be written in the same iteration before it is read, we check if the following condition holds:

$$\{(a, a, i) | a = f(i, y), i \in D_r(y)\} - \{(a, b, i) | \exists j. (a, b, i, j) \in R\} = \emptyset \quad (13)$$

If so, the array is a temporary and consequently it can be expanded.

6.2.1 Example

The following program fragment demonstrates the working of our algorithm.

```

for (int i = 0; i < 100; i++) {
    t[0] = 0;
    u[0] = 0; // S0
    for (int j = 1; j < 99; j++) {
        t[j] = t[j] * 2;    // S1
        u[j] = u[j-1] * 2;  // S2
    }
    for (int j = 1; j < 99; j++) {
        A[i][j] = t[j];
        B[i][j] = u[j];    // S3
    }
}

```

Variable `t` cannot be array expanded because within statement `S1`, the values in `t[1]`, ..., `t[98]` are loop-carried. On the other hand, variable `u` can be expanded. In `S2`, all values read in `u[j-1]` are defined in either `S0` or `S2`. Similarly, all values read in `S3` are defined by either `S0` or `S2` and are thus not loop-carried.

The array expanded code is the following:

```
for (int i = 0; i <= 99; i++) {
    t[0] = 0;
    u[i][0] = 0;
    for (int j = 1; j <= 98; j++) {
        t[j] = 2 * t[j];
        u[i][j] = 2 * u[i][-1 + j];
    }
    for (int j = 1; j <= 98; j++) {
        A[i][j] = t[j];
        B[i][j] = u[i][j];
    }
}
```

7 Affine Scheduling

Affine scheduling is the main technique of extracting resource unbounded parallelism in the R-Stream mapper. The general problem statement is as follows: given a generalized dependence graph $G = (V, E, \mathcal{C})$, compute a set of (parametric) multidimensional schedules:

$$\Theta_x : \mathbb{Z}^{d(x)+gp} \rightarrow \mathbb{Z}^{2d(x)+1}$$

for all statements $x \in V$, where $d(x)$ denotes the number of loop dimensions in the statement x .

Because of its generality, affine scheduling subsumes many aspects of traditional transformations in a classical compiler, including interchange, reversal, skewing, scaling, fusion and distribution.

We can classify the many variants of affine scheduling in the literature by the following set of criteria:

- Is there any restriction on the form of schedules?
- Does the algorithm compute parametric or non-parametric schedules?
- What sort of parallelism does it extract, i.e., coarse-grained or fine-grained, or a combination of both?
- What computation techniques are used to find the schedules?

7.1 General template of affine scheduling algorithms

All of the affine scheduling algorithms that we are examining have a similar algorithmic structure.

The general template of the algorithm is as follows:

1. Given a dependence graph G , decompose G into a set of strongly connected components (SCC) $G'_1, G'_2, G'_3, \dots, G'_n$. Process each SCC independently.
2. For each SCC, $G'_i \subseteq G$, find a one-dimensional affine schedule θ such that the maximal number of edges in G'_i can be dismissed. An edge $e \in G$ can be dismissed iff θ

$$\theta_{dst(e)}(i, y) - \theta_{src(e)}(j, y) \geq 1, \quad \forall (i, j) \in R_e(y) \quad (14)$$

Note that once θ satisfies the above condition for an edge e , subsequent dimensions of the schedule – no matter what they are – are always legal with respect to e .

3. Remove dismissed edges from each SCC G'_i , and recursively apply the algorithm to compute other (lower) dimensions of the schedule.

Different scheduling algorithms deviate from each other by how θ is computed in step (2) of the template.

7.2 Feautrier's algorithm

Feautrier's affine scheduling algorithm [Fea92a, Fea92b] uses the following to solve step (2):

2. Find θ by solving the following optimization problem:

$$\begin{aligned} \max \quad & \sum_{e \in V(G'_i)} z_e \quad \text{s.t.} \\ & \theta_{dst(e)}(i, y) - \theta_{src(e)}(j, y) - z_e \geq 0, \quad \forall (i, j) \in R_e(y), \forall e \in E(G'_i) \quad (15) \\ & 0 \leq z_e \leq 1 \end{aligned}$$

The intuitive meaning of the above optimization problem is to find a 1-D schedule θ such that the maximal number of dependence edges can be dismissed.

Note that the optimization problem above involves an infinite family of linear constraints. Using the affine form of the Farkas Lemma, we can transform the above set of constraints into a linear program (with a finite number of linear constraints).⁸

7.3 Darte and Vivien's algorithm

Instead of dealing with arbitrary dependence relations, Darte and Vivien's algorithm [DV94, DV95, DRV00] first performs *uniformization* of the dependence graph as a preprocessing step to simplify the dependence. Uniformization approximates dependence relations by transforming dependence relations into dependence vectors. The process of uniformization may also introduce *virtual nodes* in the dependence graph, which are not present in the original dependence graph.

Uniformization works as follows. Given a dependence relation R_e , we compute its Minkowski decomposition in terms of lines l , rays r and vertices v :

$$R_e = \text{lin.space}\{l_1, \dots, l_n\} + \text{cone}\{r_1, \dots, r_m\} + \text{convex}\{v_1, \dots, v_k\}$$

Let $e = (u, v)$. We then create a new virtual node w . The uniformized dependence graph contains the following new edges connecting u , v and w :

- An edge from u to w with dependence vector $\vec{0}$.
- For each $i = 1 \dots n$, one edge from w to w with dependence vector l_i and one edge with dependence vector $-l_i$.
- For each $i = 1 \dots m$, one edge from w to w with dependence vector r_i .
- For each $i = 1 \dots k$, one edge from w to v with dependence vector v_i .

See Figure 3 for an illustration of this uniformization step.

If $m = 0$ and $n = 0$, we also can omit the virtual node and create the following subgraph instead:

⁸Also, because of complementary slackness, the optimal solutions to z_e must be 0 and 1. So integer linear programming is not needed.

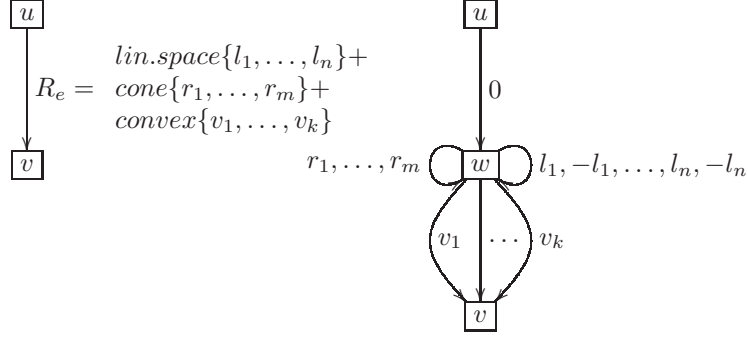


Figure 3: Uniformization of a dependence edge.

- For each $i = 1 \dots k$, one edge from u to v with dependence vector v_i .

After uniformization, step (2) in the algorithmic template is solved by computing the maximal zero-weight multi-cycle of the uniformized dependence graph via the Karp, Miller, Winograd (KMW) decomposition.⁹

Let W denote the weight matrix of the edges, and let C denote the connection matrix of the dependence graph.¹⁰ Darte et al. have shown that the following linear program can be used to solve the maximal zero-weight multi-cycle problem:

$$\begin{aligned} \min \quad & \sum_{e \in E} z_e \\ & Wq = 0, Cq = 0, q + z \geq 1 \\ & z \geq 0, q \geq 0 \end{aligned} \quad (16)$$

Let \mathcal{C} denote the maximal zero-weight multi-cycle subgraph. Then results from (16) can be interpreted as follows:

- $z_e = 0$, then e is in \mathcal{C} ,
- $z_e = 1$, then e is not in \mathcal{C} .

The dual LP of (16) can be used to compute the desired schedule θ . In Darte and Vivien's algorithm, the schedules θ order operations of a given statement into wave-fronts. This means that the schedule for statement v , θ_v , is of the form $\theta_v(i) = Xi + \rho_v$, where X is a vector common to all the linear parts of the schedule θ , and ρ_v is the offset specific to schedule θ_v .

Constraints on the dual LP can be computed using the results computed in (16):

$$e \in \mathcal{C} \implies Xw(e) + \rho_{src(e)} - \rho_{dst(e)} \geq 0 \quad (17)$$

$$e \notin \mathcal{C} \implies Xw(e) + \rho_{src(e)} - \rho_{dst(e)} \geq 1 \quad (18)$$

Solving for X and ρ above gives us the desired schedule for (2) in the algorithmic template.

⁹In a multi-cycle, an edge can be visited more than once.

¹⁰A connection matrix C is a boolean matrix such that $C_{ij} = 1$ iff edge (i, j) is present.

7.4 Lim and Lam’s affine partitioning algorithm

Lim and Lam’s affine partitioning algorithm is similar in structure as Feautrier’s algorithm, and can be thought of as a minor modification of the former to extract coarse-grained and pipelined parallelism.

- 2a. Given G' , compute its feasible schedule polyhedron:

$$P = \{\theta \mid \theta_{dst(e)}(i, y) - \theta_{src(e)}(j, y) \geq 0, \forall (i, j) \in R_e(y), \forall e \in E(G'_i)\}$$

Intuitively, P can be thought of as representing all feasible schedules which satisfy the set of dependences.

- 2b. Decompose P into a combination of lines L , rays R and vertices V , i.e., $P = lin.space(L) + cone(R) + convex(V)$. The lines represent the space of doall parallelism, while the rays represent the space of pipelined parallelism.

7.5 Summary

Algorithm	Parallelism	Restrictions	Algo. techniques	Scalability
Feautrier	fine	no	Farkas Lemma, LP	100s of stm.
Darte	fine	wavefront non-parametric	Uniformization KMW decomp., LP	100s of stm.
Lim and Lam	coarse	no	Farkas Lemma Minkowski decomp.	10 stm.

Table 1: Summary of scheduling algorithms.

Table 1 summarizes all the major variants of algorithms we covered in previous sections and how they differ from each other. Both Feautrier and Darte algorithm requires linear programming (LP) to compute step (2) in the algorithm template. The difference is in how the LPs are formulated. Feautrier’s algorithm uses the Farkas Lemma to reduce an infinite family of constraints in a finite set of constraints. This process introduces extra Farkas multipliers for each constraint in the dependence relations that must be subsequently eliminated. While Darte’s algorithm does not require Farkas multipliers, the uniformization step introduces an extra virtual node for each dependence edge in the graph. Thus both algorithms tend to introduce many extra auxiliary variables. We have found that there is not significant difference between the computation cost of the two algorithms.

Lim and Lam’s algorithm is also very similar to Feautrier’s algorithm. But instead of using LP, it uses Minkowski decomposition of polyhedra to generate

the entire solution space of feasible schedules. This gives the Lim and Lam algorithm the ability to extract both coarse-grained doall parallelism and pipelined parallelism, which neither the Feautrier and Darte algorithms can do. Unfortunately, this comes at a high price in terms of scalability. We have found that we can only scale the algorithm up to about 10 statements.

All three algorithms also suffer from the following problem: they are only concerned with extracting parallelism, with little consideration with everything else. However, two schedules with the same degree of parallelism may behave very differently in practice, because of different locality and/or “quality” of generated loops.

7.6 Our Algorithm

The affine scheduling algorithm implemented in our mapper is a combination of various ideas presented in the previous sections.

- 2a. For each SCC $G'_i \subseteq G$, compute its feasible schedule polyhedron as in the Lim and Lam algorithm:

$$P = \{\theta \mid \theta_{dst(e)}(i, y) - \theta_{src(e)}(j, y) \geq 0, \forall (i, j) R_e(y), \forall e \in E(G'_i)\}$$

We compute P by first applying the Farkas Lemma to each dependence relation R_e separately, eliminating the resulting Farkas multipliers, then joining the separate results. That is, we first compute

$$P_e = \{[\theta_{dst(e)}, \theta_{src(e)}] \mid \theta_{dst(e)}(i, y) - \theta_{src(e)}(j, y) \geq 0, \forall (i, j) R_e(y)\}$$

then combining the separate P_e relations.

- 2b. Compute $lin.space(P)$, which represents the set of doall parallelism. Suppose $P = \{x \mid Ax + b \geq 0\}$. This can be computed by simply solving for $Ax + b = 0$.
- 2c. Find a one-dimensional affine schedule such that the maximal number of edges in G'_i is as in step (2) in Feautrier’s algorithm.

The advantage of this algorithm is as follows:

- By computing P_e separately as preprocessing step, we can avoid the process of eliminating many Farkas multipliers at the same time. Also, the P_e sets may be reused in recursive invocation of the algorithm.
- Our algorithm can be used to extract coarse-grained parallelism just like Lim and Lam’s.¹¹
- Linear programming is much more scalable than Minkowski decomposition.

¹¹But not pipelined parallelism.

7.6.1 Computing “wavy” schedules

Wavefront schedules are often more desirable because they can be easily rendered in an efficient loop nest, or fit into the underlying execution model. For example, wave-front schedules are often used in systolic arrays mappings.

We can easily modify the above algorithm to compute a wave-front schedule. One possible modification is to use Darte’s KMW decomposition as a primitive in step (2) of the algorithm.

Another possible alteration — which is actually easier to implement — is to restrict the schedule θ to only wave-front schedules within the Farkas Lemma approach. Let $\theta_v(i) = \gamma i + \rho_v$. Then we can compute a wave-front only schedule using the following set of constraints:

$$\begin{aligned} \max \quad & \sum_{e \in V(G'_i)} z_e \quad \text{s.t.} \\ & \gamma i + \rho_{src(e)} - \gamma j - \rho_{dst(e)} - z_e \geq 0, \quad \forall (i, j) \in R_e(y), \forall e \in E(G'_i) \\ & 0 \leq z_e \leq 1 \end{aligned} \quad (19)$$

Another modification is to compute a wave-front schedule with the same degree of parallelism if one exists, but compute a non-wave-front schedule otherwise. We can accomplish this as follows. Let $\theta_v(i) = \gamma_v i + \rho_v$. Then we solve this LP:

$$\begin{aligned} \max \quad & \sum_{e \in V(G'_i)} z_e - N|\gamma_{dst(e)} - \gamma_{src(e)}|_1 \quad \text{s.t.} \\ & \theta_{dst(e)}(i, y) - \theta_{src(e)}(j, y) - z_e \geq 0, \quad \forall (i, j) \in R_e(y), \forall e \in E(G'_i) \\ & 0 \leq z_e \leq 1 \end{aligned} \quad (20)$$

Here, N is a sufficiently large positive constant. If a wave-front schedule exists, then the linear parts of the schedule θ_v for all v becomes identical, and the term $-N|\gamma_{dst(e)} - \gamma_{src(e)}|_1$ goes to zero.

7.7 Summary and Related Works

Traditional parallelizing compilers [AK02, Wol96] often rely on ad hoc loop transformations techniques. Such techniques are pattern matching based, and heavily dependent on the syntactic loop structure, and as such, the set of transformations and the applicability of such transformations are severely restricted. A typical phase-ordered setup is to apply unimodular transformations [WL91] to perfectly nested loops, and to apply loop fusion to sequences of loop nests.

The main weaknesses of the traditional approach are these:

- The reliance on syntax and pattern matching means that many semantically equivalent loop nests cannot be transformed because they do not fit the predefined patterns.

- The same reliance also means that each transformation is responsible for code generation, i.e., updating the program representation, which can be complex when performed as a syntactic transformation.
- The granularity is often coarser. For example, unimodular loop transformations applied to a perfectly nested loop often treat the entire body as a single statement.
- Finally, by necessity the parallelization process has to be broken down into many small transformation steps, and it can become very difficult to locate a good final solution through a series of intermediate transformations, especially if a “good” solution gauged by a combination of different and potentially conflicting measures.

In contrast, general affine scheduling techniques which works entirely in the polyhedral model suffers from none of these problems. The constraint-based approach allows us to explore the entire possible space of legal schedules, and delegate the task of solution searching to a constraint solver.

Nevertheless, there are some remaining weaknesses to our current approach which requires additional improvements to the basic scheme. The most important of these weaknesses are:

- Computationally, affine scheduling techniques are much more intensive than traditional approaches, because *(i)* the use of dependence polyhedra instead of less accurate approximations, *(ii)* the finer granularity of scheduling, *(iii)* the need to consider the dependencies of the entire program, and *(iv)* the use of linear programming and/or integer linear programming.
- Our current algorithms can only explore the space of one dimension of schedules at a time.
- Our current algorithms do not incorporate the proper set of cost metrics and can thus result in suboptimal schedules. For example, they do not directly optimize for temporal or spatial locality, or try to reduce communication.

All these issues have been addressed recently in the work of Vasilache [Vas07] on simplifying dependence relations and unified formulation of multi-dimensional schedules, and the work by Bondhugula [BBK⁺07] on scheduling with communication minimizing cost-metrics. We intend to continue improve R-Stream’s schedulers by incorporating ideas from this research.

8 Forming Kernels: Grouping and Tiling

In the past decades, micro-processor makers have worked around the increasing disparity between processing speed and memory bandwidth in several ways:

- By introducing intermediary levels of memory between the processor and the main memory. These additional levels are faster (and even more so as they are located on-chip) but smaller than the main memory. In shared memory machines, these are typically cache memories and scratch pads. In distributed memory machines, they are just local memories. Loading a memory element is costly, so once a memory element is loaded into a higher level of memory, it should be accessed as much as possible before it has to be replaced (which is bound to happen as the capacity of these memory is limited) to minimize the number of times it will have to be reloaded. The number of times a data is used once loaded into a higher level characterizes its *temporal locality*.
- Coarsening the grain of communication among these levels of memory is another way of dealing with the disparity between processing speed and memory bandwidth. With cache hierarchies, it is obtained by using cache lines, i.e., fixed amounts of contiguous elements of memory that move from one cache level to another at once. The use of cache lines virtually multiplies the memory bandwidth by c , the number of words that fit in a cache line. But, it only does so if the c words present in the cache line are all accessed by the processor before the cache line is replaced by another cache line. Once a cache line is loaded to the next level of cache, the more elements are accessed before the line is replaced, the better the performance. *Spatial locality* defines how well elements of a cache line are used before the cache line is replaced. Spatial locality applies to many different examples than cache, as in systems with virtual memory and a translation look-aside buffer (TLB), which stores virtual-physical memory page mappings in a table in a fast memory. Replacing an entry of such a table typically has a latency of hundreds of processor cycles. In distributed memory systems, direct memory access (DMA) engines also transfer blocks of contiguous memory at once. The size of the blocks is usually variable and some of the engines allow for one-time transfers of elements that are distant from each other by a constant number of words (these transfers are called *strided*, and the stride is the constant distance, usually counted in bytes). As the size of transfers is variable, an additional way of reducing the number of transfers is to transfer data through few big transfers rather than many small transfers.

Moreover, in parallel computers, communication is faster among processing elements than from a lower level of memory. Also in some architectures, it is faster, from a given processing element, to communicate data to a subset of processors (called the PE's *neighbors*) of the grid, which are physically closer or better connected.

With distributed memory systems, communications have to be explicit. Before a computation is performed, any input data that is not yet present on the processing element has to be received explicitly, and any output data needed by another processing element has to be sent explicitly. In order to make big (and few) transfers, it is necessary to consider big set of computations, for which big input and output data sets are derived.

This is realized in **R-Stream** by creating bigger computational entities, which we call **tasks**. The **R-Stream** polyhedral mapper manipulates statements, associated with their iteration domain. From this representation, in which the elementary computational entity is one iteration of one statement, there are two ways of forming bigger entity:

- by **grouping** statements, i.e., putting several statements into the same task and
- by putting several iterations of these statements into the same task, which is performed using a loop transformation called **iteration tiling**. The definition of *tiling* is overloaded in the literature, as it sometimes also means explicitly defining *data* sets that must be processed within a task (this is called *data tiling*). Since data footprints are often non-convex even for convex iteration domains, and since some data may be used by several different iterations, we chose to perform iteration tiling. Throughout this document, tiling refers to iteration tiling unless stated otherwise. In the literature, tiling is also sometimes referred to as *blocking*.

From loops scanning iterations of statements, we want to build loops scanning tasks. In the following example, **a** and **b** represent statements.

```

for i=0,n {
  for j=0,n {
    for p=0,7 {
      receive_input_data(i,j,p);
      kernel(i,j,p);
      send_output_data(i,j,p);
    }
  }
}

kernel(int i, int j, int p) {
  for it = 0,64 {
    for jt = 0, 128 {
      a(i,j,p);
      b(i,j,p);
    }
  }
}

```

Tasks are instances of a computational entity for different values of loop variables i and j . Loops i and j , scanning across tasks, are called *inter-task*, while loops it and jt , scanning iterations within a task, are called *intra-task* loops.

A **kernel** is defined by the piece of code whose instances are tasks. Kernels formed by the polyhedral mapper are a function of the inter-task loops.

For parallel target architectures, the mapper also distributes these tasks across the grid of processing elements, as described in Section 9.

In the current example, this is rendered by the p loop that would enumerate processors numbered from 0 to 7.

If possible, tasks should have the following properties:

- their data footprint, i.e., the set of data that they access, should be bounded.

In the case of distributed memory machines, this is necessary as the amount of local memory is bounded. Overflowing local memories result in runtime hardware exceptions, if not detected earlier by **R-Stream** or the low-level compiler.

With shared memory machines, bounding the data footprint of tasks avoids *capacity misses*, i.e., cache (resp. TLB) misses that are due to the set of live data not fitting in the cache (resp. in the number of memory pages referenced by the TLB).

We will see later on in this section that it is always possible to bound the footprint of a task, by bounding the number of iterations that it contains;

- the ratio between their computation time and their communication time should be greater than or equal to one. This way, communication latency can be hidden by pipelining communications with computations, by using multi-buffering (see Section 11). If the ratio is less than one, only part of the communication latency can be hidden.

It is not always possible to obtain a ratio of one. The general possibility to obtain such a ratio depends on the algorithm (algorithms for which it is possible are called “compute-bound”, as opposed to “memory-bound”), on the communication engine (bus, DMA, Ethernet cards, etc.) and on the instruction-per-count realized on the processing elements.

This section presents the problem of forming kernels, whose execution-time instances are tasks, out of loop nests. It is decomposed into the sub-problems of grouping and tiling, which we describe here and show how they are solved in the **R-Stream** polyhedral mapper.

Section 8.1 shows how grouping is performed. Section 8.2 presents the performance constraints that have to be respected for the kernel formation component to avoid negative interference with other components of the mapper. Section 8.3 shows how the tiling problem is formulated and solved. We discuss how other components of the mapper interact with the tiling/grouping component in Section 8.4, and propose future improvements in Section 8.5.

8.1 Grouping

The input of the grouping phase is the immediate output from the affine scheduling component. The original program has been restructured into a set of loop nests with explicit parallelism and with an improved degree of temporal locality, i.e., loop dimensions with parallelism are marked with *doall*, and adjacent loop nests that reference the same data are fused.

The goal of the grouping algorithm is to partition the set of statements of the GDG. The set of statements forming each element of the partition is tiled later on to form a kernel, for which operations for communicating input and output data will be generated, as well as the subsequent synchronizations.

On target machines that have a host processor in addition to a grid of PEs, the question of choosing which *side* (host or PE grid) should execute a kernel arises. Some of the kernels may have much parallelism and a computation-to-communication ratio greater than one, in which case they should be mapped onto the PE grid. This is true unless they are so small that thread startup costs become comparable with execution costs or the absolute communication granularity becomes too fine to leverage spatial locality. Also, SIMDization has a significant impact on performance of compute-bound kernels, and multi-buffering also consumes a finite amount of parallelism (as presented in Section 11). Hence, it is important that the kernel has enough parallelism to account for SIMDization and multi-buffering, in addition to parallelization across the PE grid. When statements (and their iterations) that access the same data are part of the same kernel, this data can stay at a memory level that is close to the processor. In other words, there is a potential performance gain in putting fused statements into the same kernel.

The grouping component is then responsible for putting together statements and form kernels that have all the necessary properties for their mapping on the PE grid to yield better execution time than a sequential mapping on the host processor. For the sake of conciseness, such groups of statements are called *profitable*, as mapping them to the PE grid reduces their execution time. Instead of being a binary attribute, we define the **profitability** of a kernel as a value that reflects the chances of the kernel to be profitable.

Unfortunately, profitability is not simply an increasing function of the number of statements in a kernel. Adding a statement to a kernel can make it more profitable if it shares data with the other statements of the kernel, because it increases data locality. On the other hand, if this statement does not have as much parallelism as the other statements in the kernel, and if adding this statement to the kernel would turn the execution of a loop to be sequential while it would otherwise be parallel, the profitability of the resulting kernel is likely to be lower than for the initial kernel.

A grouping that includes statements with too many sequential dimensions into a single kernel loses parallelism. On the other hand, a grouping which separates two statements that access the same data may destroy temporal locality discovered in previous phases. Thus to obtain a good grouping, two conflicting objectives must be balanced.

Our grouping algorithm is based on a profitability score assigned to one or several statements, which depends upon their number of iterations and elementary operations, upon the number of parallel loop dimensions and upon the ratio between the number of iterations and the number of accessed data.

Currently, the grouping algorithm uses the following heuristics to accomplish its task. It works in a bottom-up manner, from the innermost loops to the outermost.

1. collect the statements of the innermost loops and compute their profitability as if they were kernels.
2. Among the statements that do not belong to any kernels yet, add the statements of *most profitable* innermost loop to the current kernel to be formed.
3. Consider adding adjacent statements, i.e., statements that will be executed just before (in this case they are called “previous” statements) or just after (called “further” statements) to the current kernel. Adjacent statements are added to the current kernel if they meet some or all of the following criteria:
 - The data set written by the previous statement (resp. read by the further statement) is included in the footprint of the statements already in the kernel. This ensures that adding the statement does not entail extra communication or synchronization within the kernel.
 - Including the adjacent statements does not destroy any parallelism in the current kernel. This can be checked directly as the parallel/sequential nature of the loop dimensions of a statement are an explicit attribute of the statements.
 - if the next adjacent statement is part of a different loop than the last statement visited, two cases arise:
 - the number of iterations in the loop and its inner loops is bounded by a small constant. In this case, we can heuristically decide that the whole loop will not be tiled and we can add it to the kernel.
 - in the other case, the loop is considered tilable. If it is tiled, they cannot be fused as each tile (in other words each task) has their own memory space¹²
4. When running out of adjacent statements, go back to 2.

At the end of grouping, a kernel number (sometimes called a *mapping group* number in the code) is assigned to each statement of the GDG.

The iteration domains of the statements, now organized in groups, are then tiled. Next sections give some background on iteration tiling, discuss what we mean by tiling and which objectives and constraints we are taking into account for the tiling component.

¹²However, some communications between tasks may be avoided further by the communication generation/optimization component.

8.2 Tiling

8.2.1 Orthogonal tiling

Early literature on tiling addresses a few related problems: *(i)* to enable tiling [IT88a], *(ii)* to discover parallelism (e.g. [Fea92b, LL97]), *(iii)* to improve locality (e.g. [SL99]), and *(iv)* to reduce communications in tiled loops [BDDR94]. One main disadvantage of these techniques is that they do not preserve desirable properties such as temporal locality or parallelism, since they destroy existing schedules. In short, they are not composable with scheduling techniques. Recently, tiling techniques have appeared in the literature which attempt to find or preserve such desirable properties in one single phase [AMP00a, Gri00, BBK⁺07].

If we step back and reexamine the tiling problem, we discover a useful simplification. Tiling is usually decomposed into two sub-problems: finding a *shape* to the tile, i.e., determining how much the original iteration space is to be skewed, and finding the size of the tile. As depicted in [GFL04], the shape problem is actually a scheduling problem in disguise, hence in principle it is already taken care of by our scheduling algorithm in the mapper. Only the space dimensions can be modified without modifying locality and parallelism properties, as they do not represent time. Modifying the space dimensions modifies the binding between processor coordinates and time. Hence, this degree of freedom is naturally left for the placement component.

As a result, the shape of the tiles that we should look for is necessarily a hyper-rectangle, and the relevant tiling problem is called *orthogonal tiling*. As discussed in Section 8.3, this can be formulated as a search in which each loop of the considered loop nests is associated with at most one variable that represents its *tile size*.

Another nice example of complementarity between scheduling and orthogonal tiling is shown in [XHG05] where aggressive fusion is performed and then any dependence violation is corrected by orthogonal tiling.

8.2.2 Constraints derived from the target architecture

The tiling component of R-Stream can consider or ignore each of the following constraints.

- **The data footprint of a task must be smaller than the amount of local memory.** As explained in the introduction of the current section, this is necessary for correction on distributed memory systems and for performance on shared memory systems.
- **Tile sizes should not entail load imbalance.** Tiling reorganizes iterations so as to produce inter-task and intra-task loops. A PE coordinate is then assigned to each task by the placement component (see Section 9). In order to utilize the whole PE grid, tiling must produce at least as many tasks as there are PE in the grid. Our tiling component assigns a tile size to each loop of a statement. A loop of round-trip r that gets assigned a tile size of t is turned into an inter-tile loop of round-trip $\lfloor r/t \rfloor$ and an

intra-tile loop of round-trip t . Iterations of inter-task loops are distributed along dimensions of the PE grid. To avoid loop imbalance, a loop meant to be distributed along a dimension of the processor grid of size s should have an inter-task loop trip count of at least s . In other words, the relation $s.t \geq r$ is enforced with loops that will be used for placement.

- **Tile sizes are a multiple of the SIMD width.** SIMDization is performed downstream in the compilation process, usually by the low-level compiler. In the SIMDization process, w operations of a loop are turned into a w -wide SIMD operation, where w is the SIMD width. To facilitate this process, intra-task loops are given trip counts that are multiples of w . w is provided by the machine model. As a consequence, the number of valid tile sizes is reduced by a factor of w .
- **Tile sizes are a power of 2.** The number of valid tile sizes can also be exponentially reduced by considering exclusively tile sizes that are powers of two. As SIMD widths are powers of two in general, this constraint is compatible with the previous one.

As the tiling component is based on a search in the space of tile sizes, reducing the number of valid tile sizes has a direct impact on the scalability of the tiling component.

8.3 Formulation of the tiling problem

8.3.1 Hoisting permutable loops

Many prior approaches to tiling are either restricted to perfectly nested loops [BDRR94, IT88b], or emulated in a perfectly-nested-loop situation via *embeddings* [AMP00b, LLL01]. Embeddings turn imperfectly-nested loops into perfectly-nested loops that scan a superset of the original iterations. Extra iterations which do not belong to the original loop nests are filtered by guards. The resulting perfectly-nested embedding is then tiled uniformly.

This need for perfectly-nested loops is driven by the concept of *permutability*. We can rephrase tiling transformation as permutability transformations as follows:

- Reschedule (all) the loops into permutable loops, i.e., which can be interchanged without violating the dependences;
- Strip-mine all the loops, giving two kinds of loops (the inter-tile and intra-tile loops);
- The inter- and intra-tile loops resulting from one given loop have the same scanning directions as the original loop. Hence, they inherit their *permutability* with the other loops. All the different intra-and inter-tile loops can hence be permuted with each other. In particular, all the inter-tile loops are permutable with all the intra-tile loops. The inter-tile loops

are hoisted to the outermost loop levels and the innermost loop levels kept inside.

Unfortunately, tiling modeled on permutability becomes significantly less intuitive as soon as imperfectly nested loops are considered, as the following example demonstrates:

```
for (i=1; i <= n; i++) {
    a;
    for (j=1; j <= m; j++)
        b;
    for (j=1; j <= p; j++)
        c;
}
```

If we apply permutability to the above loop nests, the following questions arise; What does permuting loops i and j imply/mean? If hoisting the inter-tile loop i is trivial, what happens to the corresponding intra-tile loop? Should the inter-tile loop j be hoisted above statement a ?

One of the several ways of embedding this loop nest would give:

```
for (i=1; i <= n; i++) {
    for (j = 1; j <= m+p; j++) {
        if (j==1) a;
        if (1 <= j && j <= m) b;
        if (m+1 <= j && j <= m+p) c;
    }
}
```

The most obvious drawbacks of this transformation are these:

- Loss of explicit parallelism. If one of the j loops is sequential, or if there is a dependence between statement $a(i)$ and $b(i)$ or $c(i)$, embedded loop j cannot be explicitly parallel. An important part of the work of the scheduling component is then destroyed.
- Over-approximation of the footprints. A unique tile size is chosen for the j loop. Hence, b and c will have the same number of iterations in a tile. Assume that the data footprint per iteration of c is significantly larger than the one of b . As the data footprint of each tile instance has to be smaller than the size of the local memory, the tile size is limited by the data footprint of c , even though b benefits from a significantly larger tile size. b could then be too fine-grained, affecting performance.
- The problem of finding a *compact* embedding, i.e., without empty iterations, is not trivial in general. The iteration domains may have different shapes that may imply splitting the iteration domain of the statements to be embedded, increasing the number of guarded statements.

- It reintroduces predicates. The raising phase integrates affine predicates into the statement’s iteration domains, leaving us with the most precise polyhedral information possible. Predicating statements with affine guards either forces us to modify their representation to include guards or forces us to use predication and make sub-optimal mapping decisions and have a sub-optimal resulting code (as polyhedral scanning may not be able to use them explicitly, unless they are reintegrated into the domain in between).

Note that Ahmed’s embedding technique [AMP00b] is technically different from the simple code sinking presented here. It is the result of a scheduling technique where the iteration spaces of all the statements are put in a common space (the Euclidean product of the iteration spaces), and then they are more or less projected into a smaller-dimensional space and tiled. However, the fundamental idea is the same, with apparently the same consequences.

We believe the problem here comes from the way the tiling transformation is conceived, i.e., its reliance on permutability. A more convenient way of looking at tiling is presented in next section.

8.3.2 Loop sinking instead of hoisting

The approach we are using is a descendant of Feautrier-style scheduling technique [Fea92a, Fea92b], with the constraint that it cannot modify the linear properties (locality, explicit parallelism, fusion) obtained by the scheduling component. We define a statement-wise tiling as a non-linear relation that can be computed dimension by dimension. Restricting Xue’s tiling definition [Xue97] to the orthogonal case, we define an orthogonal tiling by the diagonal matrix

$$T = \begin{pmatrix} t_1 & \dots & 0 & 0 \\ 0 & t_2 & \dots & 0 \\ 0 & \vdots & \ddots & 0 \\ 0 & 0 & \dots & t_n \end{pmatrix}$$

such that each tile is a hyper-rectangle whose size is given by the diagonal elements of T and whose starting iteration I_T (also called its origin) is such that $I_T \bmod T = 0$. This is illustrated with the blue rectangles at the left of Figure 4 and can be written:

$$I = T \lfloor T^{-1} I \rfloor + I \bmod T \quad (21)$$

Unlike Xue [Xue97], who compresses the whole iteration space, we want to be (at least in theory) able to tile different statements differently. So we do not compress the statements’ iteration space, which allows us to keep them our iterations in a common space. Instead, we see tiling as *collapsing* all the iterations of R onto the iteration $I_T(R)$, as shown at the right part of Figure 4.

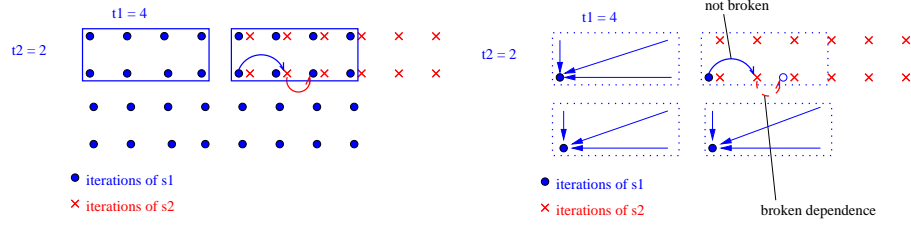


Figure 4: Tiling viewed as iteration collapsing.

Collapsing iterations means that the collection of collapsed iterations will be executed at the same iteration as the origin of the tile. This iteration will execute a loop nest that contains all the iterations belonging to the tile.

In terms of loop transformations, this can be described as a statement-wise strip-mining (giving an outer and an inner loop) followed by a loop sinking. In our method, this can be done for all the loop levels of a particular statement, and all the inner loops are then sought below all the outer loops. Using the example of the last subsection, the loops

```
for (i=1; i <= n; i++) {
    a;
    for (j=1; j <= m; j++)
        b;
    for (j=1; j <= p; j++)
        c;
}
```

would become

```
for (i=1 ; i <= n/3; i++) {
    for (it=1; it <= 3; it++)
        a;
    for (j=1; j <= m/4; j++)
        for (it=1; it <= 3; it++)
            for (jt=1; jt <= 4; jt++)
                b;
    for (j=1; j <= m/7; j++)
        for (it=1; it <= 3; it++)
            for (jt=1; jt <= 7; jt++)
                c;
```

with tile sizes: $a(3)$, $b(3, 4)$ and $c(3, 7)$. In this example the same sizes have been chosen for the shared loop (i). We shall see why this is desirable in the next section.

Now that we know what transformation we mean by tiling, let us see what are the conditions in which this transformation can be done.

8.3.3 Tilability

In this section, we show how to derive the maximum possible tile size of loops surrounding a given statement from the dependences that constrain it. The range of possible tile sizes for a loop is called its **tilability**. As we consider that a tilability of one is always feasible (which is true if the previous transformations did not violate any dependences as opposed to [XHG05]), we will most of the time make this minimum of one implicit and use *tilability* to refer to the maximum possible tile size.

Scheduling constraints holding on a pair (s_1, s_2) of statements at the k -th loop level can be of either of these forms:

$$i_{k,2} - i_{k,1} \geq \delta, (i_{k,1}, i_{k,2}) \in D_{12},$$

(loop-carried dependence) or

$$i_{k,2} - i_{k,1} = 0, (i_{k,1}, i_{k,2}) \in D_{12}$$

(dependence not carried at level k), where δ is either 1 or 0 and D_{12} is a polyhedron describing a dependence between s_1 and s_2 .

Let us treat these cases separately.

Inequalities

Tiling s_1 with a tile size of t at the k -th level would respect the dependence if and only if:

$$i_{k,2} - t \lfloor i_{k,1}/t \rfloor \geq \delta \quad (22)$$

This integer floor operation is annoying as it is non-linear; however, we can use bounds to return to a linear form:

$$i - t < t \lfloor i/t \rfloor \leq i \quad (23)$$

Proposition 8.1. *We can always tile a statement on which other statements are dependent, but which does not depend on other statements.*

Proof. According to (23), we have:

$$\begin{aligned} i_{k,2} - t \lfloor i_{k,1}/t \rfloor &\geq i_{k,2} - i_{k,1} + t \\ \Rightarrow i_{k,2} - t \lfloor i_{k,1}/t \rfloor &\geq \delta, (i_{k,1}, i_{k,2}) \in D_{12}. \end{aligned}$$

We can then always tile a statement on which other statements are dependent, but which does not depend on other statements. The distance between two dependent iterations may only increase by tiling s_1 . \square

When s_1 is dependent on other statements, things are different. The dependence is written:

$$i_{k,1} - i_{k,2} \geq \delta, (i_{k,1}, i_{k,2}) \in D_{21}.$$

Here, tiling s_1 respects the dependence if and only if:

$$t \lfloor i_{k,1}/t \rfloor - i_{k,2} \geq \delta \quad (24)$$

Equation (23) gives the following equality:

$$t \lfloor i_{k,1}/t \rfloor - i_{k,2} > i_{k,1} - t - i_{k,2},$$

which is an integer relation that we can rewrite :

$$t \lfloor i_{k,1}/t \rfloor - i_{k,2} \geq i_{k,1} - i_{k,2} - t + 1.$$

It is easy to see that

$$i_{k,1} - i_{k,2} \geq t + \delta - 1 \Leftrightarrow i_{k,1} - i_{k,2} - t + 1 \geq \delta \Rightarrow t \lfloor i_{k,1}/t \rfloor - i_{k,2} \geq \delta.$$

Then, tiling statement s_1 respects dependence (24) if

$$i_{k,1} - i_{k,2} \geq t + \delta - 1, (i_{k,1}, i_{k,2}) \in D_{21} \quad (25)$$

Note that the case $t = 1$, which means *no tiling*, works as it always respects dependences.

In general, statement s_1 will depend on several other statements, giving a set of tilability constraints:

$$P_{s_1}(t_1) = \bigcap_k i_{k,1} - i_k \geq t_1 + \delta_{1,k} - 1, \quad (26)$$

where t_1 is s_1 's tilability for the considered loop level.

Equalities

Dependences not carried at the k -th level are contained in the equation:

$$i_{k,2} - i_{k,1} = 0, (i_{k,1}, i_{k,2}) \in D_{12}$$

Tiling statement s_2 while preserving the dependence structure means that we get:

$$t_2 \lfloor i_{k,2}/t_2 \rfloor - i_{k,1} = 0, t \leq 1.$$

$t = 1$ is a trivial but least interesting solution. However, there are only two ways this equation can be satisfied:

- when $t_2 = 1$. This means that we don't tile s_1 at this level.
- when $i_{k,1}$ and $i_{k,2}$ are both multiples of t_2 .

Knowing that s_1 has to be tiled too, we come to the conclusion that for both their tile size to be greater than one, they must be a multiple of each other, i.e., they must be equal. When respecting this constraint, tilability is only limited by the dependences carried at the considered level (the inequalities).

The maximal tilability for statement s at some level k is then given by the minimum among the maximal tilabilities for all the statements that share the

same k -th level loop with s . The maximal tilabilities for individual statements is given by equation (25):

$$t_{max} = \max\{t|t \in P_s(t)\}$$

We know how to compute, for each statement of the GDG, its tilability, and we know that the tilability of a k -deep loop is the minimum among its statements' tilabilities at depth k . We will see later that in general dependences are not the only factor that limits a loop's tilability. The way statements are nested plays an important role in the formulation of the tiling problem. Next section shows how to integrate the nesting information in a natural way.

8.3.4 Tiling as a search: beta tree

The recursive structure of imperfectly nested loops naturally lends itself to a tree representation in which nodes represent loops and leaves represent statements. Figure 5 shows a loop nest and its corresponding beta-tree (in black).

Dependence-based tilabilities are represented in blue. They are propagated from the leaves (the statements, for which they were computed) to the other nodes (toward the root). Each loop inherits a dependence-based tilability from its descendants: the tilability range of a k -deep loop node is the intersection of the ranges of its descendants at level k .

Besides, loops may have a maximum trip count, and it is useless to tile a loop by a greater size than its maximum trip count. Hence, each loop has also a **trip-count tilability**, which is an upper bound on its possible tile size. It is represented in pink in Figure 5.

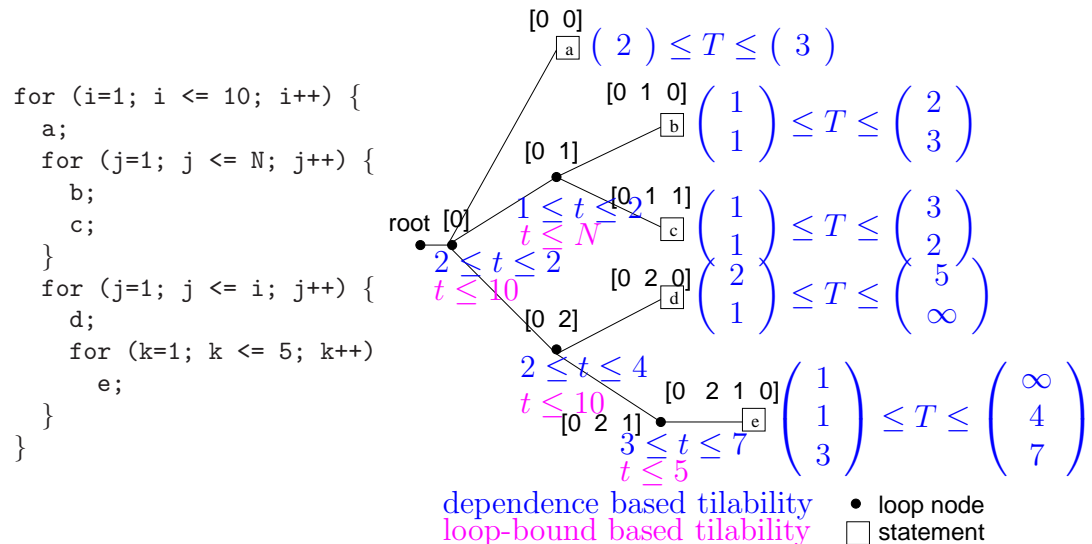


Figure 5: Tilability propagated on a β -tree.

By looking at the beta-tree, it is obvious now that the variables of our problem are the tile sizes attributed to the loops, i.e., there is one variable associated with each of the loop nodes of the beta-tree. Finding the optimal tile sizes can be formulated as a search in the space of loop tile sizes, with the following constraints:

- dependence-based tilabilities,
- loop trip-count based tilabilities
- and the constraints that come from the target machine, formulated in Section 8.2.2.

Note that the variables whose tilability is limited to one do not need to be considered as variables.

8.3.5 Consequences of our tiling paradigm

One of the well-known facts about tiling is that *a parallel loop is always tilable*. As no dependences are carried by these loops, there are only equalities at this level, which don't pose additional constraints other than the one that is explicit now due to the beta-tree: all shared loops are tiled by the same amount.

There is also an interesting relation between tiling viewed as a collapsing problem and reductions. It is well-known that, even though the iterations that compute an accumulation in a reduction are associative, a polyhedral dependence analysis over-approximates these as linear dependences, rendering them serial. This is because they cannot all be scheduled in parallel using affine schedules. A non-linear schedule (parallel prefix) has to be applied for them to become parallel. Similarly, this over-approximated (cyclic) dependence limits tilability to one if no special consideration is taken.

Let us rephrase the collapsing problem (*if n iterations can be collapsed along dimension k , then the k -depth loop is tilable*): Let a and b be two consecutive iterations (at dimension k). If n iterations can be moved between a and b , then the k -depth loop is tilable by n . As reductions make the iterations associative, we can always move as many iterations as we want between a and b . Hence, **reduction dependences can (and should) be ignored when computing tilability**. This alleviates the inherent problem of the polyhedral representation w.r.t. reductions. However, it is understood that, as long as no parallel prefix scheme is implemented, the reduction dimensions will stay sequential (but tilable).

Tiling offers a way to group a fixed-size amount of iterations of a set of statements into a tile, i.e., an atomic executional unit. On some of the target machines, these kernels have to be mapped onto the processing elements. When a statement has a tilability of one along all its dimensions (it is said to be *not tilable*), this leads us to tiles of only one iteration. When these tiles are mapped onto processing elements, it results in very fine-grained parallelism which entails extremely bad performance on coarse- and middle- grained architectures like

Cell and TRIPS. Also note that because parallel loops are always tiling, an untilable loop cannot be parallel, so to add harm to injury, these fine-grained tiles would have to be mapped sequentially.

Fortunately, there is another way to group iterations, that is always available even when the statement(s) is (are) not tiling: it is always possible to strip-mine the innermost loop and consider the resulting inner loop as the tile. This fixes the *granularity* part of the performance problem, although it does not make the code parallel.

In the current implementation of tiling, it is possible to:

- compute tiling as described in the previous sections, or base it only on the tiling made available by doall (parallel) loops and reductions. This option is called “*unbounded*” tiling.
- strip-mine the innermost loop of the non-tilable statements. We could actually extend this to any statement.

8.3.6 Implementation: generic search

We have implemented the search for an optimal tile size as a generic search with a set of:

- variables, whose values are generated according to their direct constraints (tiling, power of 2, multiple of SIMD width)
- constraints: each solution is evaluated against a set of constraints. In general, a function is evaluated and tested for positivity. If negative, it is not valid. The constraints are: footprint limit and communication minimization.
- objectives: the search has to try to maximize a combination of them. They are: data footprint maximization of a tile, maximization of the self-reuse (reuse through a fixed non-invertible access function), minimization of communications.

We are currently focusing on improving the quality of the constraint and objective functions. While they are not definite enough, it seems untimely to devise a performance specialized search algorithm. Hence, we are using a genetic algorithms search (with chromosome repair capabilities). It allows a simple trade-off between quality of the solution and scalability, by adapting the size of the populations and the number of generations in function of the size of the search space.

8.4 Interaction with other mapper components

The results of the scheduler have a great impact on what the kernel formation component (tiling/grouping) can do. The more the statements are fused, the easier it is to form big kernels. It is also well-known that the tiling of loops directly depends on the scheduler (which has to *expose* tiling).

Also, one objective and one constraint of the search are not as precise as they could be if they knew which loops were to be chosen for carrying placement:

- the loop imbalance preventing mechanism assumes that any doall loop can carry processor coordinates, which make it too conservative;
- the communication estimator is unable to make the difference between communications coming from the host and from another PE.

8.5 Future improvements

We would like to combine tiling with placement, either by computing placement before tiling, or by generating several good placement candidates to be included in the tiling search. Also, when the objective and constraint estimators are mature enough, we will work on efficient searches that use their mathematical properties (monotonicity, being a polynomial/a posynomial [RR04], etc.).

9 Processor Placement

The role of the placement component of the **R-Stream** mapper is to assign tasks to the processing elements (PE) that will execute them. For performance, the assignment should be done in such a way that the amount of parallel work is maximized and the amount of communication between processors is minimized.

Recall that, in our context, a task is a group of statements to be executed atomically on a processor. Given a statement S in a d -level loop nest, assignment means finding a processor assignment function

$$\Pi : \mathbb{Z}^d \rightarrow \mathcal{P},$$

where \mathcal{P} denotes the processor space. **R-Stream** models the target machine processors as a grid of processing elements embedded in a p -dimensional hyper-rectangle. Formally, $\mathcal{P} = \{x \mid 0 \leq x < P\}$, where P is a p -vector denoting the dimensions of the grid. For example, the TRIPS architecture can be modeled as a 2×2 grid, while the CELL architecture can be modeled as a 1-D grid with 8 processors. We identify a processing element (PE) by its p -dimensional coordinate within the grid.

As a simplification of the problem, the **R-Stream** mapper currently uses the *computer owns* rule, i.e., we assume that input data has to be in the processor's local memory at the time when the computation starts. Thus given a placement function Π , data movements between processors can be automatically inferred.

To keep the problem tractable and efficiently implementable, we currently restrict placement functions Π to *modular mappings*, which are functions of the form:

$$\Pi(x) = Ux \bmod \gamma$$

where U is unimodular transform and γ is a d -dimensional vector of the form $(\gamma_1, \dots, \gamma_p, 1, \dots, 1)$. Thus strictly speaking Π is a $\mathbb{Z}^d \rightarrow \mathbb{Z}^d$ mapping. However, since $(d-p)$ dimensions of Π are always zero, the image of Π is a p -dimensional hyper-rectangle whose dimensions are those of the PE grid. In the terminology used in HPF (High-Performance Fortran [hpf]), placement functions are restricted to a form of multi-dimensional block cyclic distribution.

9.1 Algorithm

Now, consider the following identity:

$$x = U^{-1}\Gamma.(\lfloor \Gamma^{-1}.(Ux) \rfloor) + (Ux) \bmod \gamma, \Gamma \in \mathbb{Z}^{d \times d} \quad (27)$$

where Γ is the diagonal matrix whose diagonal elements are the elements of γ .

We can proceed to a mapping from the multi-dimensional iteration space to space and time dimensions (as is done in e.g. [CF93]) by identifying time and space in equation (27):

$$t(x) = \lfloor \Gamma^{-1}.(Ux) \rfloor s(x) = (Ux) \bmod \gamma$$

So we can write a general space-time mapping for a finite hyper-rectangular PE grid as:

$$x = U^{-1}\Gamma.t(x) + U^{-1}.s(x), 0 \leq s(x) < \gamma \quad (28)$$

In our mapper, the placement phase is run after the scheduler, which is responsible for finding parallel loops with minimal communication. As we want such loops to carry placement, we are usually considering the case when U is the identity, giving the following simpler change of variable:

$$x = \Gamma.t(x) + s(x), 0 \leq s(x) < \gamma \quad (29)$$

Equations (28) and (29) define an injective $\mathbb{Z}^d \rightarrow \mathbb{Z}^{2d}$ mapping: $x \mapsto (t, s)$. However, as some of the s 's are always zero, we can eliminate them, producing an actual $\mathbb{Z}^d \rightarrow \mathbb{Z}^{d+p}$ mapping. By doing this, we introduce p variables representing the coordinates of an iteration on the processing element grid. Every iteration is hence assigned a PE coordinate.

In the terminology of traditional loop transformation, we can describe the transformation defined by equation (29) as a series of p strip-minings, where the resulting inner loops are identified with processing element coordinates.

As an example, consider a target machine with a one-dimensional grid of 8 processing elements. The following loop

```
for (i=0; i < n; i++) {
    a[i] = a[i+1] - 5;
}
```

can be strip-mined by eight using the relation $i = 8i_1 + i_2, 0 \leq i_2 < 8$, giving:

```
for (i1=0; i1 < n/8; i1++) {
    for (i2=0; i2 < 8; i2++) {
        a[8*i1+i2] = a[8*i1+i2+1] - 5;
    }
}
```

9.2 Single-Program Multiple-Data (SPMD) code generation

We can place these computations on the one-dimensional processing element grid by identifying i_2 with the one-dimensional processing elements coordinates: $i_2 = \text{proc}$. Thus, processor with coordinate proc executes the following loop nests:

```
for (i1=0; i1 < n/8; i1++) {
    a[8*i1+proc] = a[8*i1+proc+1] - 5;
}
```

Note that `proc` is not a loop index but rather an extra runtime parameter which stands for the current processor coordinate. Turning a loop variable into a processing element coordinate is only legal if the strip-mined loop is parallel, as it actually schedules them for the same value of the outer strip-mined loop (`i1` here). Turning the PE coordinates into parameters is done by the thread generation component, which also generates the loops that scan the different parallel threads.

Using the terminology of HPF, this way of doing corresponds to a cyclic distribution. However, currently in `R-Stream`, placement is applied to tiled statements, so indeed tasks are distributed across the processing elements. The loops chosen for carrying placement are inter-tile loops. It is then a form of multi-dimensional block cyclic distribution.

9.3 Minimizing communications

One of the roles of the scheduler is to make explicit parallel loops across which communications are minimal. One would then think that we can obtain a placement that minimizes communications by choosing the loop that carries the fewest communications to carry placement. This is only true if one doesn't distinguish the different kinds of communications involved. In particular, in our two-level machine model, communications can happen between processing elements or between a processing element and the host (we always consider the existence of host-side memory even when the target machine has no host processor).

With certain interconnection networks between processing elements, the proximity between two communicating processing elements directly impacts performance. Typically, neighbor-to-neighbor communications are preferred. This is really a prominent goal for fine-grain target architectures like systolic arrays and FPGAs.

A more precise goal for placement is to minimize the number of communications between the host and the processing elements and to minimize the average distance of inter-PE communications.

In the next sections, we show how to get closer to these goals.

9.4 Eliminating host broadcasts

Broadcasts from the host-side to processing elements are the most typical case where communications can be reduced further than by just choosing the best loops to carry placement.

9.5 Related works

This scheme is the opposite of the *owner computes* rule where the data are assigned to processors and the computations happen where the data is. The owner computes approach is only clearly defined in some cases. When data

necessary for a computation are spread across multiple processing elements, no processor owns all the data so ad hoc rules are used.

10 Local Memory Compaction

One of the crucial tasks of the mapper is to allocate local memory for architectures with fast but limited scratchpad memories on distributed memory architectures. When data is being migrated from one memory to another, we have an opportunity to also perform a reorganization of the data layout. The key idea that we would like to exploit is adapting the local data layout to improve storage utilization, locality of reference, or enable other optimizations. Such data layout reorganization often comes “for free” especially if it can be overlapped with computation via hardware support, such as DMA.

For example, suppose we would like to place the following loop fragment onto some remote processor:

```
double A[300][300];
for (i = 0; i < 100; i++) {
    ... = ... A[2*i+100][3*i];
}
```

By inspection, we know that only 100 elements out of 90000 of **A** are accessed within the loop. Furthermore, access to the array is not contiguous, but contains gaps, and thus will have less than optimal locality on architectures with cache. Thus keeping the original data layout (and array size) in the remote processor is extremely inefficient. As part of the process migration phase, we would like to compact the layout of the array in the local memory of the remote machine. A possible transformation is as follows:

```
double A_local[100]; // local memory
transfer A[2*i+100][3*i] to A_local[i], i = 0 ... 99
for (i = 0; i < 100; i++) {
    ... = ... A_local[i];
}
```

Transforming the reference from **A**[2*i+100][3*i] to **A_local**[i] reduces the storage requirement of local memory from 300×300 to the optimal 100 elements. Today, the above transfer can be realized efficiently with DMA hardware with strided access. Thus, performing the data rearrangement not only improves local memory utilization and locality, it can also be done very cheaply given some standard hardware support.

10.1 Motivating Examples

We now give a few more motivating examples for the local memory compaction phase.

Suppose we have the following loop fragment with a system parameter **M** and suppose we would like to map it onto a remote processor.

```

// Kernel with parameter M
for (i = 0; i <= 5; i++)
  for (j = 0; j <= 10; j++)
    for (k = 0; k <= 15; k+=2) {
      ... = A[i,i+M];
      ... = B[2*i,2*j];
      j.. = C[k/2,k/2+20];
    }

```

By examining the loop nests, we observe a few inefficiencies in memory usage. The data footprint of $A[i, i+M]$ is one dimensional but it occupies a 2-D array. For efficiency, the array A should be compacted into 1-D in local memory. Similarly, the data footprint of $C[k/2, k/2+20]$ is also one dimensional. The data footprint of $B[2*i, 2*j]$ is two-dimensional but it contains gaps, and at most 1/4 of the data is actual touched. Again, for efficiency, we would like to remove these holes when it is moved into local memory.

Consider another loop fragment below.

```

for (i = 0; i <= 5; i++) {
  for (j = 0; j <= 10; j++) {
    for (k = 0; k <= 15; k++) {
      ... = A[i+2*k, j-k+15];
    }
  }
}

```

The extents of the reference $A[i+2*k, j-k+15]$ are $[0, 35]$ and $[0, 25]$ in the first two dimensions. So we required $36 \times 26 = 936$ elements to store A in local memory. However, a change of basis to $A_1[i+2j, j-k+15]$ can reduce the storage requirements to $26 \times 26 = 676$.

Consider the following loop fragment where the loop kernel to be mapped onto a local processor is an inner loop of some larger loop nests.

```

for (i = 0; i < 100; i++) {
  // kernel starts here ...
  for (j = 3*i; j < 3*i+100; j++) {
    ... = ... A[j] ...;
  }
  // end of kernel
}

```

Note that the extent of $A[j]$ is $[0, 400)$, but within each iteration of i the kernel uses only 100 elements. If we use a naive local memory allocation scheme, we would require the same number of elements in the local memory. Clearly, the problem is that the data footprint of $A[j]$ is a function of the outer loop

index i . By shifting the reference locally to $A_local[j-3*i]$ we can reduce the storage need to the optimal 100.

Consider a similar loop fragment as the previous one. In this fragment, we have an extra system parameter N .

```
for (i = 0; i < N; i++) {
    // kernel starts here ...
    for (j = 3*i+N; j < 3*i+100+N; j++) {
        ... = ... A[j] ...;
    }
    // end of kernel
}
```

This example is very similar to previous loop fragment. However, note that we cannot allocate A locally in a bounded amount of storage if we don't readjust the indices on A , because the variable N is unbounded. However, it is also obvious that we only require 100 elements in the local memory, because each iteration of i only 100 elements of A is actually referenced. This shows that in the presence of parameters, reindexing is necessary.

Next, consider the following matrix multiply loop nests where M , P and N are system parameters. We would like to map the three innermost loops onto a remote processor.

```
for (i = 0; i <= -1 + M; i += 10)
    for (j = 0; j <= -1 + P; j += 10)
        for (k = 0; k <= -1 + N; k += 10)
            // Kernel starts here
            for (l = i; l <= min(-1 + M, 9 + i); l++)
                for (m = j; m <= min(-1 + P, 9 + j); m++)
                    for (n = k; n <= min(-1 + N, 9 + k); n++)
                        C[l, m] = C[l, m] + A[l, n] * B[n, m];
            // Kernel end here
```

In this example, the data footprints of A , B , C are all 10×10 within the l, m, n loops. We can achieve the mapping by allocating three local arrays A_l , B_l and C_l , and rewrite the global references to local references as follows:

- $A[l, n]$ to $A_l[-i+l, -k+n]$,
- $B[n, m]$ to $B_l[-k+n, -j+m]$, and
- $C[l, m]$ to $C_l[-i+l, -j+m]$.

We now move on to two examples with more complex tiled iteration spaces. The first example is the following:

```

for (i = 0; i <= 9; i += 2)
  for (j = max(-1, -9 + i); j <= min(4, 3 + i); j += 2)
    // Kernel starts here
    for (k = max(1, i, i - j);
         k <= min(4 + i - j, 1 + i, 9); k++)
      for (l = max(-i + j + k, 1);
           l <= min(4, 1 - i + j + k); l++)
        A[k, 2*l] = A[k, 2*l-2] + A[k-1, 2*l-2];
    // Kernel ends here

```

To obtain a legal local memory allocation, we allocate a local memory array A_l of size 3×7 . Then we reindex the global references to local references as follows:

- $A[k, 2*l]$ to $A_l[k-i+1, 2*l-2*j+2]$,
- $A[k, 2*l-2]$ to $A_l[k-i+1, 2*l-2*j]$, and
- $A[k-1, 2*l-2]$ to $A_l[k-i, 2*l-2*j]$.

The second example is a tiled stencil loop nests:

```

for (i = 0; i <= -1 + N; i += 3)
  for (j = 0; j <= -1 + N; j += 4)
    // Start of kernel
    for (k = i; k <= min(2 + i, -1 + N); k++)
      for (l = j; l <= min(-1 + N, 3 + j); l++)
        A[l,k] = (B[l-1,k] + B[l+1,k] + B[l,k-1] + B[l,k+1]) / 4;
    // End of kernel

```

To obtain a legal local memory allocation, we allocate two local arrays A_l and B_l of sizes 4×3 and 6×5 respectively. Then the global references are rewritten to the local references as follows:

- $B[l-1,k]$ to $B_l[l-j,k-i+1]$,
- $B[l+1,k]$ to $B_l[l-j+2,k-i+1]$,
- $B[l,k-1]$ to $B_l[l-j+1,k-i]$, and
- $B[l,k+1]$ to $B_l[l-j+1,k-i+2]$.

10.2 Algorithm

The general problem statement can be stated simply as follows. We are given a set of loop nests with parametric affine array references $A_1[f_1(x)], \dots, A_n[f_n(x)]$. The result of local memory compaction is a mapping of these global references to new local references:

$$A_i[f_i(x)] \mapsto A'_i[g_i(x)], i = 1 \dots n$$

where A'_i are new arrays to be allocated in the local memory of the remote processor.

To restrict ourselves to only “efficiently” executable transformations, the access functions g_i must be affine. Thus the transformation from $f_i \mapsto g_i$ is also affine¹³.

Our algorithm contains the following three basic steps:

- Partition the set of references so that related references are grouped and allocated together.
- For each group of referenced decided in step (1), perform algebraic simplification via Hermite decomposition [Sch86].
- For each group of references, perform geometric rearrangement via uni-modular reindexing.

The next sections describe these steps in detail.

10.3 Group related references

Our algorithm allocates different arrays independently. References to the same array are generally allocated together if they overlap. Non-overlapping references are placed into distinct allocation groups to minimize their interference.

To see why the choice of groups is important, consider the following loop nests from the innermost loops of tiled LU decomposition¹⁴

```
float A[256][256];
doall (l=128*j+16*P; l <= min(-i+254,128*j+16*P+15); l++)
    doall (m = 16*k; m <= min(-i+254, 16*k+15); m++)
        A[1+i+m][1+i+1] -= A[1-i+m][i] * A[i][1+i+1];
```

All three references to the array A happen to be disjoint, i.e., they access disjoint areas of the array when the loops. Thus we can allocate these references separately. For example, we can transform the above loop nests into the following inner loops using new local variables A_2, A_3, A_4:

```
float A_2[16][16]; // a triangular subregion of A
float A_3[16];     // a column of A
float A_4[16];     // a row of A

// DMA code omitted
```

¹³In principle, it is possible to reduce memory usage by considering other types of transformations, such as piece-wise affine functions, or modular mappings. However, the use of such transformations may cause degradation in performance in many common situations, because these indexing functions are more expensive to compute.

¹⁴Variable i, j and k are outer loop indices, and the variable P is a parameter standing for the current processor.

```
doall (l = 0; l <= min(15, -i-128*j-16*P+254); l++)
  doall (m = 0; m <= min(-i-16*k+254, 15); m++)
    A_2[m][l] -= A_3[m] * A_4[l];
```

Note that if all three references are to be allocated together as a single unit, we must map the three references to the same local array. Since only affine transformations are allowed, the amount of storage required is 256×256 , i.e., we require the same amount of local memory as global memory.

On the other hand, it is clear that it is not always optimal to group overlapping references together. For example, consider the following inner loop fragment¹⁵

```
double A[100][100];
for (j = 0; j < 100; j++) {
  .. = A[i][j] * A[j][i];
}
```

The two references $A[i][j]$ and $A[j][i]$ overlap when $i=j$. However, if we allocate the references together, it is impossible to reduce the local memory usage using only affine transformations. This is because the data footprint of the two references (a cross) is a 2-dimensional set, while the data footprints of the individual references are both 1-dimensional.

In order to compute better allocations in situations like this, our algorithm will first estimate how much overlapping is in the references. If the references are read-only, and if overlapping is a tiny percentage of the overall area, our algorithm will split the references into distinct groups. In the above example, our algorithm will generate the following local memory allocation. Note that the center element of the data footprint, $A[i][i]$, has been *replicated* and put into the locations $A_1[i]$ and $A_2[i]$:

```
double A_1[100];
double A_2[100];

Transfer A[i][j] to A_1[i], i = 0 ... 99
Transfer A[j][i] to A_2[i], i = 0 ... 99

for (j = 0; j < 100; j++)
  ... = A_1[j] * A_2[j];
```

10.4 Hermite Decomposition

The purpose of Hermite decomposition is to reduce the dimension of the reference to the actual geometric dimension of the data footprint. In addition, if the

¹⁵Variable i is an outer loop index.

access pattern contains strides, this step removes these strides in the resulting local references.

The technique is as follows. Given an affine function $f(x, y)$ on loop indices x and parameters y , we first decompose it into the sum of $g(x) + h(y)$, where $g(x)$ is a linear function on x and $h(y)$ is an affine function on y . Function $g(x)$ can be decomposed into $g(x) = HU$, where $H = [H'0]$ is the Hermite Normal Form of g and U is unimodular matrix. Let $U = \begin{bmatrix} U_1 \\ U_2 \end{bmatrix}$ where $HU = H'U_1$. We can then generate the following mapping from global to local indices:

$$f(x, y) \mapsto U_1 x$$

10.4.1 Example 1

For example, suppose we are given the following loop nests:

```
double A[300][300];
for (i = 0; i < 100; i++) {
    ... = ... A[2*i+100][3*i];
}
```

The access function is $f(i) = [2i + 100, 3i]$. Rewriting this in matrix form, we have:

$$\begin{aligned} f(i) &= \begin{bmatrix} 2 \\ 3 \end{bmatrix} [i] + \begin{bmatrix} 0 \\ 1 \end{bmatrix} [1] \\ &= \begin{bmatrix} 2 \\ 3 \end{bmatrix} [1][i] + \begin{bmatrix} 0 \\ 1 \end{bmatrix} [1] \end{aligned}$$

Thus $H = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$ and $U = [1]$, which gives the following global to local mapping:

$$[2i + 100, 3i] \mapsto [i]$$

10.4.2 Example 2

The example below contains three references, each of which is to be treated separately:

```
float A[256][256];
doall (l=128*j+16*P; l <= min(-i+254, 128*j+16*P+15); l++)
    doall (m = 16*k; m <= min(-i+254, 16*k+15); m++)
        A[1+i+m][1+i+l] -= A[1-i+m][i] * A[i][1+i+l];
```

The three references are:

$$\begin{aligned} f_1(l, m, i, j) &= [1 + i + m, 1 + i + l] \\ f_2(l, m, i, j) &= [1 - i + m, i] \\ f_3(l, m, i, j) &= [i, 1 + i + l] \end{aligned}$$

Note that the outer loop indices i, j are treated as parameters. We can decompose the references as follows:

$$\begin{aligned} f_1(l, m, i, j) &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} l \\ m \end{bmatrix} + \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} l \\ m \end{bmatrix} + \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \\ f_2(l, m, i, j) &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} l \\ m \end{bmatrix} + \begin{bmatrix} -1 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} l \\ m \end{bmatrix} + \begin{bmatrix} -1 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} l \\ m \end{bmatrix} + \begin{bmatrix} -1 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \\ f_3(l, m, i, j) &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} l \\ m \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} l \\ m \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} l \\ m \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} \end{aligned}$$

From the above decompositions, we can obtain the following global to local memory mappings:

$$\begin{aligned} A[1 + i + m][1 + i + l] &\mapsto A_1[m][l] \\ A[1 - i + m][i] &\mapsto A_2[m] \\ A[i][1 + i + l] &\mapsto A_3[l] \end{aligned}$$

10.5 Unimodular Reindexing

The final step of unimodular reindexing reorients a skewed data footprint so that the resulting access pattern is more rectangular than before. The problem

can be stated as the following optimization problem. Given

1. An array \mathbf{A} in d dimensions,
2. A set of system parameters $y = [y_1, \dots, y_k]$, and
3. A parametric data footprint set $D(y)$ for a group of references to array \mathbf{A} .

We want to find:

1. The dimensions of the local array $S : \mathbb{Z}^d$,
2. An affine function $L : \mathbb{Z}^k \rightarrow \mathbb{Z}^d$ which represents the lower bound of the local array as a function of the parameters y , and
3. An unimodular matrix $U : \mathbb{Z}^{d \times d}$, which represents the skewing to be done in the original footprint.

We can succinctly phrase this optimization problem as follows:

$$\begin{aligned} & \min \prod_i S_i \\ & L(y) \leq Ux < L(y) + S, \quad \forall x \in D(y) \\ & S : \mathbb{Z}^d \\ & L : \mathbb{Z}^d \rightarrow \mathbb{Z}^d \\ & U \text{ unimodular} \end{aligned}$$

We can interpret the above set of constraints in the following manner. The product $\prod_i S_i$ stands the total size of the local array. Minimizing $\prod_i S_i$ thus minimizes the total amount of storage required. The set $\{z \in \mathbb{Z}^d \mid L(y) \leq z < L(y) + S\}$ represents the bounding box of the local array. Finally, we allow the data footprint to be automatically “rectangularized” by a unimodular transformation U .

Given S , L and U , we can generate code for the local array as follows:

- The local array is given the array dimensions S_1, \dots, S_d .
- The global array element x is moved to the local array element $Ux - L(y)$.

10.5.1 Solving the optimization problem

The above optimization problem contains non-linear constraints, non-linear optimization objective and integer constraints. Currently, we perform the following steps to solve this optimization problem:¹⁶

- We apply Farkas Lemma to remove the universal quantification in $\forall x \in D(y)$. The result is a finite set of constraints.
- Instead of solving for all dimensions all at once, we perform a linearization of the constraints and generate a sequence of optimization problems, one for each dimension.

¹⁶The details of these steps are beyond the scope of this report.

- Each set of subproblem reduces to a small integer linear programming problem which can be solved using standard solving in milliseconds.
- The result of the optimization problems are combined. We iterate these steps if the current solution proves to be unsatisfactory.

10.6 Generating bulk communication

One of the tasks of the local memory compaction phase is to insert communication code to transfer data between global and local memories. In order to ensure communication overhead is minimized, only *bulk* communication operations are generated, and these are inserted only at the boundaries of the innermost kernels, such that there is no communication operations within a kernel. During communication insertion we assume an asynchronous communication model with the following primitives at our disposal:

- Asynchronous communication initialization operations such as **put** and **get**. These operations initiate an asynchronous transfer of a region of data from one processor to another. These (abstract) operations have the form

```
get A[f(x,y)] from B[g(x,y)] for x ∈ D(y) tag t;
put A[g(x,y)] to B[f(x,y)] for x ∈ D(y) tag t;
```

where A is a local array, B is a global array, y are the system parameters, and (f, g, \mathcal{D}) together describe the set of elements to be transferred. The semantics of the **get** operation is identical to the following loop nests:

```
for x ∈ D(y)
  A[f(x,y)] = B[g(x,y)];
```

where the **put** is identical to the following:

```
for x ∈ D(y)
  B[f(x,y)] = A[g(x,y)];
```

- A **wait t** operation which blocks the execution of a thread until a group of initiated communications have completed. Groups of communication operations are associated with a tag t .

It should be stressed that the above primitives are abstract in nature, and do not have to correspond directly to actual target machine primitives. In Section 12 we shall describe how we map these primitives into lower level DMA operations.

The following example illustrates how communication operations are inserted by our mapper. Suppose we would like to map the following matrix multiply onto a distributed memory machine with 8 distributed processor elements:

```

float A[1024][1024];
float B[1024][1024];
float C[1024][1024];
for (int i = 0; i <= 1023; i++) {
    for (int j = 0; j <= 1023; j++) {
        for (int k = 0; k <= 1023; k++) {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}

```

After affine scheduling, tiling and processor placement, we can obtain the following single-program, multiple data (SPMD) loop nests, where the newly introduced parameter P ranges from 0 to 7 and stands for the current processor id.

```

float A[1024][1024];
float B[1024][1024];
float C[1024][1024];
for (int i = 0; i <= 31; i++) {
    for (int j = 128 * P; j <= 128 * P + 127; j++)
        for (int k = 32 * i; k <= 32 * i + 31; k++)
            C[j][k] = 0;
    for (int j = 0; j <= 15; j++)
        for (int k = 128 * P; k <= 128 * P + 127; k++)
            for (int l = 32 * i; l <= 32 * i + 31; l++)
                for (int m = 64 * j; m <= 64 * j + 63; m++)
                    C[k][l] = C[k][l] + A[k][m] * B[m][l];
}

```

The above loop nests operate on arrays A, B, and C, which are located in the memory of the host processor. In order to produce a distributed memory mapping, we perform the local memory compaction optimization described in this section and produces the following loop nests. New local variables A_l, B_l and C_l are introduced and placed within the memories of the remote processor units:

```

local float A_l[128][64];
local float B_l[64][32];
local float C_l[128][32];
float A[1024][1024];
float B[1024][1024];
float C[1024][1024];
for (int i = 0; i <= 31; i++) {
    for (int j = 128 * P; j <= 128 * P + 127; j++)

```

```

        for (int k = 32 * i; k <= 32 * i + 31; k++)
            C_l[j - 128 * P][-32 * i + k] = 0;
    put C_l[j][k] to C[j + 128 * P][32 * i + k],
        for 0 <= j <= 127, 0 <= k <= 31 tag 1;
    wait 1;
    for (int j = 0; j <= 15; j++) {
        // fetch submatrices from A, B and C
        get B_l[k][l] from B[64 * j + k][32 * i + l]
            for 0 <= k <= 63, 0 <= l <= 31 tag 0;
        get A_l[k][l] from A[k + 128 * P][64 * j + l]
            for 0 <= k <= 127, 0 <= l <= 63 tag 0;
        get C_l[k][l] from C[k + 128 * P][32 * i + l]
            for 0 <= k <= 127, 0 <= l <= 31 tag 0;
        wait 0;
        // kernel
        for (int k = 128 * P; k <= 128 * P + 127; k++)
            for (int l = 32 * i; l <= 32 * i + 31; l++)
                for (int m = 64 * j; m <= 64 * j + 63; m++)
                    C_l[k - 128 * P][-32 * i + l] +=
                        B_l[-64 * j + m][-32 * i + l] *
                        A_l[k - 128 * P][-64 * j + m];
        // write back a submatrix of C
        put C_l[k][l] to C[k + 128 * P][32 * i + l]
            for 0 <= k <= 127, 0 <= l <= 31 tag 1;
        wait 1;
    }
}

```

10.7 Related Work

Our algorithmic framework is based mostly on the work of Schreiber et al. [SC04]. Our work generalizes theirs in different ways:

- Our unimodular reindexing algorithm works for the full polyhedral model with arbitrarily shaped parametric iteration domains. In contrast, previous work only considers perfectly nested loops with non-parametric rectangular iteration spaces.
- Our algorithm correctly deals with domains with strides by first extracting the lattice within the domains.
- Our algorithm deals with non-convex data footprints by
 - Splitting disjoint sets into distinct data references
 - Splitting regions with small overlaps

- Our algorithm also automatically generates asynchronous DMA communication operations to transfer data between local and global memories. Multi-buffering is used to overlap communication and computation.

11 Multi-buffering

The mapping in the previous section can be further improved by allowing communication and computation to be overlapped in time via the use of double-buffering or multi-buffering schemes. The idea of double-buffering is as follows: data needed for iteration $i + 1$ of the computation kernel is prefetched in one buffer while we are executing iteration i using the other buffer. Similarly, we can overlap writing data back with prefetching data and with computations by delaying the write-back by one iteration and adding yet another buffer. Data for iteration $i + 1$ is prefetched into one buffer while computation kernel for iteration i is executed using the second buffer and output data from iteration $i - 1$ is sent back from the third buffer.

Triple buffering makes it possible to fetch, execute and write back at the same time, when the target machine allows for it, i.e., when it has full-duplex asynchronous communication capabilities. Double buffering is useful when communications are half-duplex (and asynchronous). Using more buffers (which is called *multi-buffering*) tends to smoothen communication times, but it also reduces the size of data sent and received. Hence it reduces the granularity of parallelism.

Let B be the number of buffers used, a the number of iterations separating the fetching of input data from the execution of the kernel that uses it, and l the number of iterations between the execution of a kernel and the sending of data it has produced. The relation between a , l and B is given by

$$a + l \leq B - 1$$

There are only two useful cases:

- the equality, which maximizes the positive impact of covering communications with computations;
- and the case when $a + l = B - 2$, in which case one buffer is not used for either communication or computation. Instead, it contains the results of computation $i - (B - 1)$. In the case when there is a lot of reuse between tasks i and $i - (B - 1)$, output data from task $i - (B - 1)$ can be copied locally instead of going back and forth to the host or another PE. This optimization is especially relevant when the reuse is not perfect, i.e., when there is not a data set that is used and/or defined and that is always the same across task iterations (for instance across iterations of the innermost inter-task loops). In this latter case, the perfectly reused data set should be “promoted”, i.e. it should live constantly in the PEs’ local memory as long as it is fully reused.

11.1 Multi-buffering with loop interchange

Our communication insertion algorithm performs these steps:

1. If we are performing B -buffering where B is the number of buffers, then we readjust the upper limit of local memory by a factor of $1/B$ to ensure that there is enough local memory. Each local array is given B versions in the mapped layout.
2. We perform loop interchange in the outer loop nests surrounding the kernels in order to ensure the loop nests immediately surrounding a kernel has no loop-carried dependences. Hence, what happens (sending or receiving data, working on a present data set) in one of the buffers is independent of what is done in any other buffer, which makes it legal to have send and receive operations executed concurrently with tasks.
3. When $B > 1$, we perform *loop-shifting* to shift `get` operations ahead a iterations.
4. If $B > 2$, the write-back step can also be pipelined. If so, we perform loop-shifting to shift the `wait` operation, which waits for the completion of the `put` operations back l iterations, with $l = B - a - 1$.

Note that loop interchange and loop shifting can be expressed as simple transformations in the polyhedral representation (see Sections 2 and 4).

For example, using the above steps, we can obtain the following pipelined execution of matrix multiply (double-buffering is used.)

```

local float A_l[2][128][64];
local float B_l[2][64][32];
local float C_l[2][128][32];
float A[1024][1024];
float C[1024][1024];
float B[1024][1024];
for (int i = 0; i <= 15; i++) {
    if (i == 0) {
        for (int j = 0; j <= 32; j++) {
            if (j >= 1) {
                swap C_l[0] and C_l[1];
            }
            if (j <= 31)
                for (int k = 128 * P; k <= 128 * P + 127; k++)
                    for (int l = 32 * j; l <= 32 * j + 31; l++)
                        C_l[0][k - 128 * P][-32 * j + l] = 0;
            if (j >= 1) wait 1;
            if (j <= 31)
                put C_l[0][k][l] to C[k + 128 * P][32 * j + l]
                for 0 <= k <= 127, 0 <= l <= 31 tag 1;
        }
    }
}
for (int j = -1; j <= 32; j++) {

```

```

    if (j <= 31 && j >= 0) {
        wait 0;
        swap A_1[0] and A_1[1];
        swap B_1[0] and B_1[1];
        swap C_1[0] and C_1[1];
    }
    if (j <= 30) {
        get B_1[1][k][l] from B[64 * i + k][32 + 32 * j + 1]
            for 0 <= k <= 63, 0 <= l <= 31 tag 0;
        get A_1[1][k][l] from A[k + 128 * P][64 * i + 1]
            for 0 <= k <= 127, 0 <= l <= 63 tag 0;
        get C_1[1][k][l] from C[k + 128 * P][32 + 32 * j + 1];
            for 0 <= k <= 127, 0 <= l <= 31 tag 0;
    }
    if (j <= 31 && j >= 0) {
        for (int k = 128 * P; k <= 128 * P + 127; k++)
            for (int l = 32 * j; l <= 32 * j + 31; l++)
                for (int m = 64 * i; m <= 64 * i + 63; m++)
                    C_1[k - 128 * P][-32 * j + 1] +=
                        B_1[-64 * i + m][-32 * j + 1] *
                        A_1[k - 128 * P][-64 * i + m];
    }
    if (j >= 1) wait 1;
    if (j <= 31 && j >= 0) {
        put C_1[0][k][l] to C[k + 128 * P][32 * j + 1]
            for 0 <= k <= 127, 0 <= l <= 31 tag 1;
    }
}
}

```

A few clarifications are due in order to explain the above loop nests:

- the outermost doall loop was interchanged to the third loop level (k in the code).
- Two copies of local arrays are reserved for the variables A_1 , B_1 , C_1 . We represent these extra copies as an extra dimension in the array indices.
- In the actual generated code, two pointers are used to perform swapping. The swapping operation can be implemented by simply swapping these two pointers.
- When $B > 2$, the swapping operation is replaced by a more general buffer rotation operation.
- The prologue and epilogue stages of the pipeline are embedded within the loop nests and are selectively enabled and disabled using the proper

guards. While the placement of these guards seems complex, they are indeed automatically generated in the final code generation phase of the mapper (See Section 15.)

11.2 Multi-buffering with loop jamming

Let B be the chosen number of buffers. The loop interchange technique presented in last section puts a doall loop as innermost inter-task (inter-tile) loop, so that tasks executed consecutively are independent of each other, and so their buffers can be processed (sent, received, computed on) concurrently. Doing this is sufficient, but not necessary. The necessary condition is that any task k is independent of tasks whose buffer will be processed concurrently with the buffers of task k . These tasks are tasks $(k - B + 1)$ to $(k + B - 1)$. In other words, the necessary condition for multi-buffering to be legal is that two dependent tasks are separated by $B - 1$ independent tasks. The basic idea behind the multi-buffering technique presented in this section is to insert $B - 1$ independent iterations between dependent iterations of the innermost inter-task (inter-tile) loop level.

This new method is motivated by several disadvantages of the multi-buffering technique presented in the previous subsection.

- As buffer rotation is performed through a run-time pointer there is no direct correspondence between loop indices and buffer indices. Hence, analysis of dependences involving arrays in local memory (so in buffers) cannot be stated as a function of loop indices or parameters. As a consequence, dependence analysis is not precise for those data.
- The performed loop interchange modifies (destroys) linear properties obtained by the scheduling component: grain of parallelism and locality of data accesses.
- One doall loop is selected to scan sequential but independent iterations that periodically use the buffers. Hence, all the independent iterations of this loop are sequentialized, while only B sequential independent iterations are necessary between two dependent iterations, for all the B send, receive and execute operations to be independent. In other words, the amount of parallelism sacrificed for multi-buffering with the loop interchange method equals the trip count of the doall loop that is interchanged, while only a factor of B parallel iterations would be necessary.

The new multiple buffering scheme allows to obtain multi-buffering in the same cases in which the old one can, but without the issues of the latter.

The old scheme ensures a dependence distance of at least B by having the innermost loop nest scan independent iterations. This actually ensures an dependence distance.

The nice properties of the new scheme come from the fact that it introduces a dependence distance of exactly B iterations between iterations that work on

the same buffer. As there are only B (or $B - 1$, depending on how we define the difference) independent tasks executed between two originally successive tasks, the buffer of the originally previous task is still available on the processor and hence data locality of the original program may be preserved.

Let us show what is done on the following example (which has a similar structure as matrix multiply). For readability, intra-task loops are represented with the function calls `init` and `update`. Loops scanning independent iterations are noted `doall` instead of `for`.

```
doall (i=0; i<= n; i++) {
  doall (j=0; j<= n; j++) {
    init(i,j);
    for (k=0; k<= n; k++) {
      update(i,j,k);
    }
  }
}
```

Imagine that we choose to use the parallelism present in the j loop to set a quadruple-buffering scheme that receives two buffers ahead. In the original code, there is a dependence distance of 1 in the innermost loop.

We can strip-mine the j loop by 4 (this corresponds to the change of variable). Let us first assume that n is a multiple of 4.

```
doall (i=0; i<= n; i++) {
  doall (j'=0; j'<= n/4; j'++) {
    doall (j''=0; j''<=3; j''++) {
      init(i,j', j'');
      for (k=0; k<= n; k++) {
        update(i,j',j'',k);
      }
    }
  }
}
```

and sink the inner strip-mined loop (j'') down to the innermost loop level:

```
doall (i=0; i<= n; i++) {
  doall (j'=0; j'<= n/4; j'++) {
    doall (j''=0; j''<=3; j''++) {
      init(i,j, j'');
    }
    for (k=0; k<= n; k++) {
      doall (j''=0; j''<=3; j''++) {
```

```

        update(i,j',j'',k);
    }
}
}
}

```

Since the iterations of j'' are independent, the dependence distance between consecutive iterations of k is of exactly 4 iterations. Now it remains to generate the communications, to shift the receive operations ahead by 2 iterations and to shift completion of the send operations by 1 iteration. Let us look at the code with communications:

```

doall (i=0; i<= n; i++) {
    doall (j'=0; j'<= n/4; j'++) {
        doall (j''=0; j''<=3; j''++) {
            init(i,j, j'');
            for (k=0; k<= n; k++) {
                doall (j'''=0; j'''<=3; j'''++) {
                    receive(A,B,C, i,j',j'',k);
                    wait_receive(A,B,C, i,j',j'',k);
                    update(i,j',j'',k);
                    send(C, i,j',j'',k);
                    wait_send(C, i,j',j'',k);
                }
            }
        }
    }
}
}

```

11.2.1 Shifting problem

Now we need to shift the “receive” operations by 2 and the “send completion” operations by one.

We could do this naively by a change of variables on j'' : $j''_{recv} = j'' - 2$ and $j''_{send} = j'' + 1$.

But the problem in doing so is that, by construction, j'' is a very short loop. Hence, the pipelining effect obtained by shifting the communication operations is very small, and the resulting performance improvement insignificant.

Example

```

doall (i=0; i<= n; i++) {
    doall (j'=0; j'<= n/4; j'++) {
        doall (j''=0; j''<=3; j''++) {
            init(i,j, j'');

```

```

    }
    for (k=0; k<= n; k++) {
        doall(j''-2; j''<=4; j''++) {
            if (j'' <=1) {
                receive(A,B,C, i,j',j''+2,k);
            }
            if (j''>=0 && j''<=3) {
                wait_receive(A,B,C,i,j',j'',k);
                update(i,j',j'',k);
                send(C, i,j',j'',k);
            }
            if (j''>=1) {
                wait_send(C, i,j',j''-1,k);
            }
        }
    }
}

```

The pipeline is initiated, enters a very short steady state and finishes, for each value of k . Indeed send, receive and computations are covered only across two iterations for each value of k .

For multi-buffering to be worth, what we really want is to pipeline communications with computations of tasks across a long innermost loop. To do this, we merge loops k and j'' into a bigger innermost loop that carries inter-task dependence of distance B before shifting the communication operations. This is done by a specific projection that is a particular case of polyhedral flattening [Claus00]. Let us call k' the variable for this new loop, obtained by the following change of variables:

$$k' = 4k + j''$$

If the only constraints on k are $0 \leq k \leq 3$, the order of execution of the iterations of the resulting k' loop is exactly the same as the one of loops (k, j'') . Note that k and j'' can be expressed in function of k' as

$$k = \lfloor \frac{k'}{4} \rfloor, j'' = k' \bmod 4$$

In our example, this gives:

```

doall (i=0; i<= n; i++) {
    doall (j'=0; j'<= n/4; j'++) {
        doall (j''=0; j''<=3; j''++) {
            init(i,j, j'');
        }
        for (k'=0; k'<= 4n+3; k'++) {

```

```

        receive(A,B,C, i,j',floor(k'/4), k'%4);
        wait_receive(A,B,C, i,j',floor(k'/4), k'%4);
        update(i,j',floor(k'/4), k'%4);
        send(C, i,j',floor(k'/4), k'%4);
        wait_send(C, i,j',floor(k'/4), k'%4);
    }
}
}

```

Now we can shift the receive operation two iterations ahead, giving:

```

doall (i=0; i<= n; i++) {
    doall (j'=0; j'<= n/4; j'++) {
        doall (j''=0; j''<=3; j''++) {
            init(i,j, j'');
        }
        for (k'=-2; k'<= 4n+3; k'++) {
            if (k' <=4n-2) {
                receive(A,B,C, i,j',floor((k'+2)/4), (k'+2) %4);
            }
            if(k' >=0) {
                wait_receive(A,B,C, i,j',floor(k'/4), k'%4);
                update(i,j',floor(k'/4), k'%4);
                send(C, i,j',k' % 4,floor(k'/4));
                wait_send(C, i,j',k' %4,floor(k'/4));
            }
        }
    }
}
}

```

Integer divisions and modulo operations are generally to be avoided, because they are not linear so they are not easily represented within the polyhedral model. Also, their computation may be costly at execution time, unless they are powers of two, like in this example. Besides, we would like to know explicitly which buffers are accessed, and the natural way of doing that is to have an affine relation between the loop indices and the buffer number. Removing a modulo 4 operation suggests the following change of variable (corresponding to a strip-mining of width 4):

$$k' = 4c + d, 0 \leq d \leq 3$$

The resulting code is:

```

doall (i=0; i<= n; i++) {
  doall (j'=0; j'<= n/4; j'++) {
    doall (j''=0; j''<=3; j''++) {
      init(i,j, j'');
      for (c = -1; c<=n; c++) {
        for (d=max(0, -2-4c), d<=min(4n-4c+3); d++) {
          if (4c+d <=4n-2) {
            receive(A,B,C, i,j',c,floor((d+2)/4),(d+2) %4);
          }
          if (4c+d >=0) {
            wait_receive(A,B,C, i,j',c, d);
            update(i,j',c,d);
            send(C, i,j',c,d);
            wait_send(C, i,j',c,d);
          }
        }
      }
    }
  }
}

```

The modulo operation can be automatically removed with the following identity:

$$(d+2) \bmod 4 = \begin{cases} d+2 & \text{for } d \leq 1 \\ d+2-4 = d-2 & \text{for } d \geq 2 \end{cases}$$

To remove the modulo/integer part, the loop index set for the receive operation should then be split into $d \leq 1$ and $d \geq 2$. The resulting code is:

```

doall (i=0; i<= n; i++) {
  doall (j'=0; j'<= n/4; j'++) {
    doall (j''=0; j''<=3; j''++) {
      init(i,j, j'');
    }
    for (c = -1; c<=n; c++) {
      for (d=max(0, -2-4c), d<=min(4n-4c+3); d++) {
        if (4c+d <=4n-2 && d<=1) {
          receive(A,B,C, i,j',c,d+2);
        }
        if (4c+d <=4n-2 && d>=2) {
          receive(A,B,C, i,j',c+1,d-2);
        }
        if (4c+d >=0) {
          wait_receive(A,B,C, i,j',c, d);
          update(i,j',c,d);
          send(C, i,j',c,d);
        }
      }
    }
  }
}

```

```

        wait_send(C, i, j', c, d);
    }
}
}
}
}

```

The same process (which can also be described as flattening, shifting, de-flattening and index set splitting) is also applied to the send completion operations, except that we shift them one iteration later.

The resulting code is as close as possible to the original code in the sense that:

- it is scanned in a similar way: along (i, j, k) , although the number of iterations of j have been divided by 4 and the number of iterations of k multiplied by 4 (and k is strip-mined). Hence, it preserves properties (locality, parallelism) brought by any previous transformation based on the linear characteristics of the program.
- as two dependent iterations of the original program get separated by exactly as many iterations as there are buffers, the values carried by the dependences are still present in a given buffer when they are needed. Locality properties of the original program are then preserved, even though it is on a set of smaller data sets. This is also true for locality between different statements that are close but not in the same loop, like `init` and `update` in the example: the distance introduced is small enough to avoid spill-and-fill between their executions.
- Also, no buffer rotation is necessary as the buffers are explicitly identified (with loop index d).

We have seen this technique with an example. In the next section, we summarize the whole process and generalize it to any number of buffers.

11.2.2 New multi-buffering scheme: summary

Let B be the number of buffers we want to set and a the number of iterations we want the receive operations to be executed ahead of time. The process of setting a multi-buffering scheme can be described with the following algorithm:

Select a doall loop to carry the buffer dimension. This loop may carry a lot of communications, which would turn inter-processor communications into inter-buffer communications (i.e., reads and writes internal to the processing elements). Let us call j the corresponding variable in the iteration space. Proceed to the following change of variables:

$$j = Bj' + j'', 0 \leq j'' \leq B - 1$$

and sink the j'' loop as the innermost inter-tile loop (in the case of imperfectly nested loops, this is a sink to the innermost inter-tile loops for each of its tiles). Let k be the innermost loop level before the sinking of j'' . The following operations are done on all the kernels. Generate asynchronous communications the simplest way, i.e., so that send and receive operations for an iteration are directly followed by their wait for completion (called `wait_send` and `wait_receive` here).

1. Move the receive operations ahead by a (inter-tile) iterations by doing the following change of variables (that includes index set splitting)

$$\begin{pmatrix} k \\ j'' \end{pmatrix} = \begin{cases} \begin{pmatrix} c \\ d+a \end{pmatrix} & \text{for } d \leq B-a-1 \\ \begin{pmatrix} c+1 \\ d+a-B \end{pmatrix} & \text{for } d \geq B-a \end{cases}$$

where c and d are the new innermost loop variables.

2. Move the send completion (`wait_send`) operations later in time by $(B-a-1)$ iterations by the following change of variables

$$\begin{pmatrix} k \\ j'' \end{pmatrix} = \begin{cases} \begin{pmatrix} c-1 \\ d+a+1 \end{pmatrix} & \text{for } d \leq B-a-2 \\ \begin{pmatrix} c \\ d+a-B+1 \end{pmatrix} & \text{for } d \geq B-a-1 \end{cases}$$

Because we don't actually perform the polyhedral flattening, there is no need for the trip count of the j loop to be a multiple of B . When it is not, d inherits the additional constraints that appear on j'' .

The main advantages of this method are as follows:

1. there is a direct affine relation between loop index d and the buffer numbers. Buffer numbers can be represented as an additional dimension to the arrays, whose index is then an affine expression of d ;
2. hence, there is no explicit buffer rotation to be performed, i.e., inserted by the mapper;
3. only a finite amount of doall parallelism is turned into pipeline parallelism between communications and computations, as opposed to a whole degree of parallelism with the loop interchange method.

11.3 Hierarchical multi-buffering

When target machines have more than two levels of explicitly managed memory, communications have to be issued at every level. In this document, a level is called "*higher*" when it is closer to the PEs. In the classical machine model, the host is lower than the PEs. An index is assigned to levels, from zero (for the lowest) up. Let L be the number of levels.

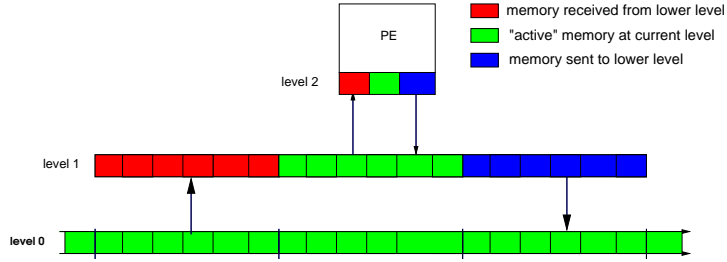


Figure 6: Hierarchical multi-buffering.

When memory level l can communicate data to level $l + 1$ and level $l - 1$ asynchronously and at the same time, it makes sense to overlap both types (directions) of communications. Hence, in addition to multi-buffering between levels $L - 1$ (PEs) and $L - 2$, it is profitable to use multi-buffering between levels $L - 2$ and $L - 3$, and so on. The minimal number of buffers still depends on the communication capabilities between levels $L - 2$ and $L - 3$. The communication layer must be able of asynchronous communication for any multi-buffering to be useful. If it is half-duplex, two buffers are necessary, three if it is full-duplex. A hierarchical multi-buffering with three levels is shown in figure 6.

In hierarchical mappings, a set of tasks at level l can be seen as one bigger task at level $(l - 1)$. Optimal communication granularity is also typically bigger between levels $(l - 2)$ and $(l - 1)$ than between levels $(l - 1)$ and l . Hence, it usually makes sense to communicate bigger data sets between levels $(l - 2)$ and $(l - 1)$, and to have fewer occurrences of these communications.

12 DMA Optimization

The Direct Memory Access (DMA) optimization component of the polyhedral mapper is responsible for turning the abstract communication commands into an optimal set of DMA transfer commands supported by the underlying target architecture. Recall from Section 10.6 that the abstract communication commands have the following forms:

```
get A[f(x)] from B[g(x)] for  $x \in \mathcal{D}(y)$  tag  $t$ ;  
put A[g(x)] to B[f(x)] for  $x \in \mathcal{D}(y)$  tag  $t$ ;  
wait  $t$ 
```

Here, `get` and `put` operations are issued to transfer an arbitrary region of memory from one array to another. All operations are tagged with an integer tag, and can be blocked until completion when a `wait` operation with the corresponding tag is executed.

While there is a wide array of different DMA architectures in the market, we can safely assume that the following list of characteristics hold for all of them:

- All DMA transfer operations can be asynchronously issued, i.e., DMA operations can be executed without delaying computation code, thus allowing communication and computation to overlap in time.
- Multiple DMA operations may be issued in parallel, up to some limit.
- One-dimensional strided accesses are possible.
- While many DMA engines impose size and alignment restrictions on the transfer parameters, we can assume that data structures allocated by the compiler and by the mapper can be made naturally aligned. Thus part of the alignment restrictions can be trivially satisfied at compile time.

We are interested in mapping the above DMA operations into one of more of the following 1-D strided DMA primitives:

```
dma_get{src=s,dst=t,  
        count=n,bytes=b,  
        srcStride=ss,  
        dstStride=ds,  
        tag=tag}  
dma_put{src=s,dst=t,  
        count=n,bytes=b,  
        srcStride=ss,  
        dstStride=ds,  
        tag=tag}  
dma_wait{tag=tag}
```

where the semantics of `dma_get` in terms of pseudo code is

```

dma_get{src=s,dst=t,
        count=n,bytes=b,
        srcStride=ss,
        dstStride=ds,
        tag=tag} =
for (i = 0; i < count; i++, s += ss, d += ds) {
    memcpy(d, s, b);
}

```

The semantics of `dma_put` can be stated similarly.

In terms of execution cost of DMA operations, we assume the following is true:

- There is a substantial startup cost for each DMA operation.
- There is a transfer cost for each byte in the message.

The actual startup cost and transfer rate is parameterized in the machine model of the target machine. However, given such constraints, we can formulate the DMA optimization problem as follows:

- Minimize the total number of DMA commands issued.
- Minimize the total number of bytes transferred.

12.1 Example

Let us now clarify the DMA generation problem with an example. Suppose we are given the following communication commands embedded in a loop nest:

```

double A[128][128];
double B[128][128];
double A_l[16][16];
double B_l[16];
double C_l[16][16];

for (i = ...) {
    for (j = ...) {
        ...
        get A_l[k][l] from A[16 * P + k][16 * i + l]
            for 0 <= k <= 15, 0 <= l <= 15 tag 0;
        get B_l[k] from B[16 * j + k][16 * i + k]
            for 0 <= k <= 15 tag 0;
        get C_l[l][k] from C[16 * P + k][16 * i + l];
            for 0 <= k <= 15, 0 <= l <= 15 tag 0;
    }
}

```

```

        wait 0;
        ...
    }
}

```

In each iteration of i and j , we fetch a rectangular 16×16 block of A into A_1 , a diagonal segment of B into B_1 and another rectangular 16×16 block of C into C_1 , but with its rows and columns permuted, i.e., a *corner turn*.

One possible mapping is as follows:

```

for (i = ...) {
    for (j = ...) {
        ...
        dma_get{src=&A[16*P][16*i], dst=&A_1[0][0],
                count=16, bytes=16*sizeof(double),
                srcStride=128*sizeof(double),
                dstStride=16*sizeof(double), tag=0};
        dma_get{src=&B[16*j][16*i], dst=&B_1[0],
                count=16, bytes=sizeof(double),
                srcStride=128*sizeof(double),
                dstStride=sizeof(double), tag=0};
        for (l = 0; l < 15; l++) {
            dma_get{src=&C[16*P][16*i+l],
                    dst=&C_1[l][0],
                    count=16,
                    bytes=sizeof(double),
                    srcStride=128*sizeof(double),
                    dstStride=sizeof(double),
                    tag=0
                };
        }
        dma_wait{tag=0};
        ...
    }
}
}

```

We can interpret the DMA mapping as follows:

- For array A , we use a strided DMA command to transfer 16 rows of A , with each block having 16 elements. The result is compacted into the array A_1 .
- For array B , we use a strided DMA command to transfer a diagonal segment of A into A_1 . The strided DMA command transfers 16 blocks each of 1 element long. The source stride is staggered so that the diagonal is transferred.

- For array **C**, we issue 16 strided DMA operations to transfer **C_1**. Note that a single strided DMA operation is insufficient because it is a corner turn.
- Finally, we issue the **dma_wait** command to block until all DMA operations have completed. Of course, it is possible to overlap communication and computation by delaying the wait operation further. For simplicity, we treat this as orthogonal issue in this section. (See Section 11 for details.)

12.2 Algorithm

The DMA optimization algorithm implemented in the **R-Stream** mapper currently defaults to a C style storage layout order, i.e., row major order. When compiling a language with a different storage layout order, suitable modification can be made to the parameterizable machine model.

We currently optimize each communication command separately. Given a communication command,

get/put $A[f(x, y)]$ **from/to** $B[g(x, y)]$ **for** $x \in \mathcal{D}(y)$ **tag** t ;

The algorithm partitions $\mathcal{D}(y)$ into a set of polyhedra whose union contains the integer points of $\mathcal{D}(y)$. Such polyhedra are defined by

$$M(x_0, y) = \mathcal{D}(y) \cap S(x_0, y), x_0 \in \mathcal{D}(y) \cap \mathbb{Z}^{d+p+1}$$

where d is the number of variables, p is the number of parameters, and $S(x_0, y)$ is the $(d - c)$ -dimensional subspace of \mathbb{Q}^d that is spanned by a set of $(d - c)$ vectors $S = (s_1, \dots, s_{d-c})$ and contains point x_0 . It is always possible to find an integer unimodular matrix V whose rightmost column vectors span $S\mathbb{Q}^{d-c}$ (proof uses Hermite normal form). Let \mathcal{D}' be the image of \mathcal{D} by V . The $(d - c)$ last variables of \mathcal{D}' scan the image of the M 's. Within a data set to be communicated, there is no legality constraint on the order in which the data should be communicated, so any unimodular V is legal.

Let $M'(x'_0, y)$ be the image of $M(x_0, y)$ through V . Each distinct $M'(x'_0, y)$ is meant to be communicated at once by a single strided communication operation. Let $a(x) = a.x + \alpha$ and $b(x) = b.x + \beta$ be the storage functions for arrays A and B .

The exact data set M' can be communicated at once using a communication operation if

$$m_a(x'_1, \dots, x'_c) = \{(a_{c+1}, \dots, a_d) \cdot (x'_{c+1}, \dots, x'_d)^T \mid x' \in D'(y)\}$$

and

$$m_b(x'_1, x'_c) = \{(b_{c+1}, \dots, b_d) \cdot (x'_{c+1}, \dots, x'_d)^T \mid x' \in D'(y)\}$$

are each defined by exactly one \mathbb{Z} -polyhedron for $x' \in \mathcal{D}'$. Note that this is always true in the case when $d - c = 1$ and $f(x, y)$ and $g(x, y)$ are invertible. Hence, a heuristic for minimizing the number of communication operations is

to find S that minimizes c . This would not take into account the number of data along S but only the number of data dimensions in each communication operation.

m_a is communicated onto m_b for all the valid integer values of (x'_1, \dots, x'_{c-1}) in \mathcal{D}' . This set of valid values is basically the projection of \mathcal{D}' on the subspace of variables (x'_1, \dots, x'_{c-1}) , which defines the iteration domain of the communication operation that we are building.

If m_a and m_b are each one \mathbb{Z} -polyhedron, a base address and stride is computed for both sides:

$$\text{base}(A, (x'_1, \dots, x'_c)^T) = (a_1, \dots, a_c)(x'_1, \dots, x'_c)^T + \min(m_a(x_1, \dots, x_c)) + \alpha$$

$$\text{base}(B, (x_1, \dots, x_c)^T) = (b_1, \dots, b_c)(x'_1, \dots, x'_c)^T + \min(m_b(x_1, \dots, x_c)) + \beta$$

The strides are given by the linear part of the supporting lattices of m_a and m_b , which are constant across the values of y and $(x_1, \dots, x_c)^T$ (up to the “center” of the lattice, which is meaningless w.r.t. strides and is accounted for in the base address).

The number of packets in the message is given by the Ehrhart polynomials of m_a and m_b . If there is a bijection between m_a and m_b , this number is the same. If not, it means that there are gather or scatter operations or both, and the number of transferred data may be ill-defined. If there is either a scatter or a gather involved, the number of packets is the maximum between the Ehrhart polynomials of m_a and m_b . Instead of computing and comparing the Ehrhart polynomial explicitly, we can take the Ehrhart polynomial of the reference that has the highest rank.

The size of a packet is set to the size of the array element (assumed to be the same on both sides).

12.3 Special cases

12.3.1 Big packets

When the size of m_a and m_b is independent of (x_1, \dots, x_{c-1}) , and when they are both contiguous (their supporting lattice is \mathbb{Z}), each m_a and m_b can be considered as one packet and further data dimensions can be aggregated into the communication operation. In this case, we rerun the same process on the iteration domain, by considering what is currently a message as a data element.

12.3.2 Strides not allowed on one side

When the targeted communication layer does not allow strided communication on one side, the only desirable option is to build messages made of contiguous data on that side. Without loss of generality, let us assume that the side for which strided communication is not allowed is the side of array A .

Hence, the column-vectors of S are chosen such that the last dimension of fV^{-1} are the $(d - c)$ last canonical vectors.

Note that for performance considerations, communicating contiguous data on one side is usually a good choice, even when there is no restriction on the communication layer used.

12.3.3 Strides not allowed on any sides

When strides are not allowed on both sides, unless there exists c and S such that the rightmost column-vectors of both fV^{-1} and gV^{-1} are the last canonical column-vectors, direct transfer is impossible.

In this case, it is necessary to copy output data into a buffer with the right data layout before sending it and/or after receiving it. In this report, we call this buffered communication mode “copy-and-communicate”.

Since we usually want to minimize the amount of data to communicate, the sending side copies its output data into a buffer, the buffer is sent (and received) as a whole. The data is laid out as needed by the recipient (which is computed by the local memory optimization component). The abstract communication operations are turned into copies to the local buffer, and a `buffer_send()` is issued. Normally, the recipient does not need to modify the layout of the incoming data, as it was laid out optimally for it by the sender.

12.3.4 Bijection between both sides

When the elements to be communicated between **A** and **B** are related by a bijection (one-to-one relation)¹⁷ it is simpler to work on one of the images, i.e., $f(D(y), y)$ or $g(D(y), y)$. They are generally defined by a union of \mathbb{Z} -polyhedra, so it is always possible to form communication commands with $d - c = 1$ for each element of this union.

12.4 Further optimization

12.4.1 Simplifying the data transfers by transferring more

In some cases, communicating more data than necessary allows to produce simpler communication operations. It is important, however, to not update a value invalidly, which would break the program’s semantics.

Examples where enlarging the data set seems profitable:

- interleaved messages whose union is a \mathbb{Z} -polyhedron
- data sets with small data holes
- other than that, when the data set to be communicated can be over-approximated by a bounding hyper-rectangular box, and when the density of data in the box is high enough. This is subjective and could be set to 50% for instance.

¹⁷which is the case currently for the abstract communications provided by the LMO component

12.4.2 Optimizing for data transfer size

Some communication engines (for instance DMA hardware engines) reach optimal performance when the message has a certain size. Also, certain communication libraries do not allow message sizes to exceed a given size.

In either case, tiling¹⁸ can be applied to \mathcal{D}' , in such a way that the number of data in the tile equals the optimal or maximal number of data. Typically, if the trip count of the innermost loop level is too short, more tiling levels will be used. Note that because the data can be sent in any order, tiling copy operations is always legal.

m_a and m_b are then formed with the “intra-tile” loops. Again, if their size is constant, the optimization used in section 12.3.1.

12.4.3 Optimizing for memory banks

Many hardware memories are organized in banks. Data transfers are generally faster when successive data transfers are issued to different memory banks. Hence, in addition to forming messages with certain properties, it is desirable to schedule the communication operations in such a way that consecutive operations are issued on different memory banks.

Scheduling data transfers for one communication operation

For this, the stride e at loop level c must be greater than $(k \times \text{bank_size} \times (\text{nb_banks} + 1)) \leq e \leq ((k + 1) \times \text{bank_size} \times \text{nb_banks})$, where k is existentially quantified. As the full access function is defined by $a(f(Vx', y))$, i.e., $a.f.Vx'$, when x'_c is incremented, the corresponding stride in A is given by the c^{th} column-vector of afV . Formally, the constraint is then

$$e = (afV.c)$$

$$\exists k : (k \times \text{bank_size} \times (\text{nb_banks} + 1)) \leq e \leq ((k + 1) \times \text{bank_size} \times \text{nb_banks})$$

Other constraints are to be taken into consideration: V has to be full-rank. In particular, it has to be independent of the subspace S spanning the message. Also, V has to be unimodular and its last $d - c$ column-vectors must span S .

There is no obvious way to build the set of solutions to all these constraints, in particular the unimodularity constraint. Hence, a constructive solution seems more appropriate.

First, we can often eliminate k by fixing it to the maximum size of a message¹⁹ and computing k_{\min} , the minimum k that makes $(k \times \text{bank_size} \times (\text{nb_banks} + 1))$ bigger than the maximum transfer data set size.

Then, let T defined by $V = (T \mid S)$. We must define v'_c , the new v_c , as a combination of the vectors of V such that the resulting matrix, let us call it V' , is still unimodular. In other words, we must have $V' = VU$, where U is

¹⁸this is iteration tiling, not data tiling, as we are tiling copy operations

¹⁹In general, determining this requires combining polynomial upper bounds of Ehrhart polynomials and Bernstein polynomials. We can ensure a non-parametric upper bound by limiting the maximum size of a transferred data set as in section 12.4.2

integer unimodular. As we don't change the other column-vectors of V , U is of the form:

$$U = \begin{pmatrix} 1 & 0 & \dots & u_{1,c} & \dots & 0 & 0 \\ 0 & 1 & \dots & \vdots & \dots & 0 & 0 \\ \vdots & & \ddots & \vdots & & & \vdots \\ \vdots & & & u_{c,c} & & \vdots & \\ \vdots & & & \vdots & \ddots & & \vdots \\ 0 & 0 & \dots & \vdots & \dots & 1 & 0 \\ 0 & 0 & \dots & u_{d,c} & \dots & 0 & 1 \end{pmatrix}$$

We can ensure unimodularity of U by fixing $u_{c,c} = 1$. So the problem becomes: build a linear combination v'_c of the vectors of V such that

$$(k_{min} * bank_size * (nb_banks + 1)) \leq afV u_{\cdot,c} \leq ((k_{min} + 1) * bank_size * nb_banks),$$

with $u_{c,c} = 1$, which is a system of constraints with $(d - 1)$ free variables. Any solution is valid, even though to get clean code we will also try to minimize the coefficients of $u_{\cdot,c}$.

One can also set e to a fixed value like *bank_size* and make the constraint set simpler. However, there might be no solution to it.

This technique is valid and has solutions if and only if the data set to be communicated spans more than one memory bank.

Interleaving data transfers for different communication operations

Another solution is to interleave transfers of data that belong to different memory banks. This is done by first computing the communication operations and their iteration domain, and then fusing loops of the same dimension and, if possible, of “similar” shapes, in order to limit the amount of control overhead that such fusion would entail.

12.5 Implementation

The current implementation deals with the case when there is a bijection between both sides, is limited to $d - c = 1$ and chooses S such that communicated data are contiguous on the PE side.

13 Register Tiling

In some cases, the low level compiler does not optimize the PE code for pipelining, SIMDization and register pressure. For such targets, R-Stream starts its *register tiling* component (also called “jamming” as its result is similar to the well-know unroll-and-jam transformation, without any unrolling) .

The role of this component is to expose as innermost loops:

- a loop scanning s doall-parallel iterations, where s is the SIMD/vector width of the target PE. These iterations are meant to be SIMDized, i.e., turned into target-specific vector intrinsics.
- a loop scanning p pipelinable iterations, where p is large enough to fill the processor pipeline and small enough to avoid register spilling. Depending on the capabilities of the target low-level compiler and PEs, these iterations may scan either doall-parallel iterations or pipelinable iterations, like for instance accumulations or reductions. doall-parallel iterations often use more registers than iterations with more reuse, like accumulations.

As tilability analysis was already performed in the tiling component, we know from which loops can be used to for the loops for SIMDization and for pipeline filling. Our representation keeps track of doall loops additionally, so there is a clear distinction can be made between pipelinable loops (which are the tilable loops) and doall-parallel loops.

The loops are formed through a transformation that we called “jamming” as it resembles unroll-and-jam, without the unrolling. It is equivalent to a strip-mining followed by a sinking of the inner strip-mined loop.

The loops to be used for SIMD and pipeline loops are chosen according to several criteria:

- maximal temporal locality (reuse) for the pipeline loop
- maximal spatial locality for both loops
- alignment and data layout constraints, depending on the target architecture, especially its SIMD constraints,
- if there is a loop whose iterations are parallel without communications, it will be used for SIMD preferably.

The SIMD width of the PEs is given (in bits) as part of the mapper’s machine model.

13.1 Implementation

There are 1-dimensional and 2-dimensional versions of the register tiling component. The 1-dimensional component currently exposes doall iterations for SIMDization, while the 2-dimensional exposes both types of innermost loops.

14 Array Contraction

Array contraction is the opposite of the array expansion optimization described in Section 6. Whereas array expansion tries to remove false dependences in a program by increasing the number of storage locations, array contraction tries to reduce the amount of storage required by collapsing multiple array locations into a single one.

A classic example of array contraction is the replacement of an array reference of $A[i]$ into an array of length N by the reference $A'[i \% L]$, which indexes into a bounded buffer of length L .

The basic idea is that different data that are not live at the same time can share the same memory location (i.e., the same array element).

Here is a very simple example of array compaction:

```
for (i=0; i<n; i++) {  
    a[i] = f(i);  
    b[i+1] = a[i];  
}
```

Here, each element of a is live during only one iteration. Hence, all the elements of a can use the same memory location. In other words, a can be contracted into an array of one element or equivalently a scalar. Let us call this scalar $a2$:

```
for (i=0; i<n;i++) {  
    a2 = f(i);  
    b[i+1] = a2;  
}
```

This process is similar to but not identical to the local array compaction process described in Section 10. Local array compaction does not attempt to assign multiple in-use locations in the global array to the same array location in the local array. Array contraction, on the other hand, does, using liveness information as a guide.

The array contraction problem can be stated as follows: given a program together with its schedule (space-time mapping), compute for each array A a *contraction function* f_A . The contraction functions are applied to all references of A . For example, given a reference $A[g(x)]$, the transformed reference is $A[f(g(x))]$. Generally speaking, we are interested in finding “easy-to-compute” non-injective functions f to serve as contraction functions. Of course, in the degenerate case we may not be able to perform any contraction without violating the semantics of the program. In such cases we can simply choose f to be the identity function.

14.1 Lattice based framework

Our array contraction algorithm is based on the lattice based framework of Darte, Schreiber and Villard described in [DSV03, DSV04, DSV05]. This frame-

work computes modulo mappings of the form

$$f(x) = (Mx) \hat{\text{mod}} b \quad (30)$$

where $(a_1, \dots, a_n) \hat{\text{mod}} (b_1, \dots, b_n)$ is interpreted to mean $(a_1 \hat{\text{mod}} b_1, \dots, a_n \hat{\text{mod}} b_n)$ and $a \hat{\text{mod}} b$ is defined as follows:

$$\begin{aligned} a \hat{\text{mod}} b &= a \bmod b && \text{if } b \neq 0 \\ a \hat{\text{mod}} b &= a && \text{if } b = 0 \end{aligned}$$

Thus modulo mappings also include linear mappings as a degenerate case.

Darte et al.'s framework has the following basic steps:

1. For each array \mathbf{A} , compute the *conflict set* \mathcal{C} of \mathbf{A} . The conflict set \mathcal{C} is a polyhedral set such that $(i, j) \in \mathcal{C}$ implies that locations i and j cannot be mapped to the same location without violating the semantics of the program. Conflict set takes the role of
2. Given \mathcal{C} , compute its difference set $\mathcal{D} = \{i - j \mid (i, j) \in \mathcal{C}\}$.
3. Compute the Minkowski decomposition of \mathcal{D} , i.e., $\mathcal{D} = L + V$. Note that because \mathcal{D} is symmetric, the decomposition does not contain rays. From the decomposition:
 - (a) The lines L describes the dimensions of \mathbf{A} which cannot be contracted.
 - (b) The polytope V describes dimensions which can be contracted. We find a hyper-parallelepiped P which includes V . Intuitively P describes the amount of total storage which are required to store the contractible dimensions of \mathbf{A} .

From these two components we can compute the contraction function f .

By varying the accuracy of these steps we can derive various algorithms, some trading compilation speed for compactness of the contraction, and some trading compactness for compilation speed.

14.2 Algorithm

The preliminary array contraction algorithm implemented in the R-Stream mapper currently follows these steps:

- compute an approximation of the conflict set;
- compute a linear array contraction function.

We compute the conflict sets using a simple approximation rather than expensive array dataflow analysis. The latter requires solving parametric integer programming problems. The approximation assumes that all array locations are live from the time when the first operation assigns it to the time when the last operation reads it.

Formally, we can phrase the approximation as follows. Suppose that there is a dependence between statements S and T . Let the dependence polyhedron \mathcal{R}_{ST} denote this dependence, i.e., $(i, j) \in \mathcal{R}_{ST}$ implies $\langle S, i \rangle$ depends on $\langle T, j \rangle$. Let $A[f(x)]$ and $A[g(x)]$ denote the references in S and T respective which induce the dependence. And let Θ_S and Θ_T denote the schedules of S and T . Then define

$$\mathcal{C} = \left\{ (a, b) \mid \begin{array}{l} (i, j) \in \mathcal{R}_{ST} \\ (i', j') \in \mathcal{R}_{ST} \\ a = f(i), b = f(j) \\ \Theta_S(i) \succ \Theta_T(j') \\ \Theta_S(i') \succ \Theta_T(j) \end{array} \right\} \quad (31)$$

For simplicity, we also restrict the contraction functions that we compute to only linear contractions, i.e., all functions are of the form $f(x) = Mx$ rather than $f(x) = (Mx) \bmod b$. This allows us to stay in the pure polyhedral model.²⁰ We shall relax this restriction in the future when we have more experience with uses of array contraction.

14.3 Example

The following is a simple example of array contraction in action. We are given a simple 3-stage filter. Array \mathbf{C} is used for both input and output, while arrays \mathbf{A} and \mathbf{B} are used to hold temporary data.

```
for (int i = 0; i <= N; i++) {
  for (int j = 0; j <= M; j++) {
    A[i][j] = f(C[-2 + i][1 + j]);
  }
  for (int j = 0; j <= M; j++) {
    B[i][j] = g(A[i][1 + j], A[i][j], C[-1 + i][j]);
  }
  for (int j = 0; j <= M; j++) {
    C[i][j] = h(B[i][j], A[i][2 + j], A[i][1 + j]);
  }
}
```

The filters are written as three separate loop nests. Using loop fusion, we can improve the spatial and temporal locality of the program. The result is the following loop nests:

```
if (M >= 0) {
  for (int i = 0; i <= N; i++) {
    for (int j = -2; j <= min(M + -2, -1); j++) {
      A[i][2 + j] = f(C[-2 + i][3 + j]);
    }
  }
}
```

²⁰Modulo indexing functions are outside of the polyhedral model.

```

    for (int j = 0; j <= M + -2; j++) {
        A[i][2 + j] = f(C[-2 + i][3 + j]);
        B[i][j] = g(A[i][1 + j], A[i][j], C[-1 + i][j]);
        C[i][j] = h(B[i][j], A[i][2 + j], A[i][1 + j]);
    }
    for (int j = max(0, M + -1); j <= M; j++) {
        B[i][j] = g(A[i][1 + j], A[i][j], C[-1 + i][j]);
        C[i][j] = h(B[i][j], A[i][2 + j], A[i][1 + j]);
    }
}
}

```

Finally, array contraction can be applied to further reduce the memory usage of the loop nests.

```

if (M >= 0) {
    for (int i = 0; i <= N; i++) {
        for (int j = -2; j <= min(M + -2, -1); j++) {
            A[2 + j] = f(C[-2 + i][3 + j]);
        }
        for (int j = 0; j <= M + -2; j++) {
            A[2 + j] = f(C[-2 + i][3 + j]);
            B = g(A[1 + j], A[j], C[-1 + i][j]);
            C[i][j] = h(B, A[2 + j], A[1 + j]);
        }
        for (int j = max(0, M + -1); j <= M; j++) {
            B = g(A[1 + j], A[j], C[-1 + i][j]);
            C[i][j] = h(B, A[2 + j], A[1 + j]);
        }
    }
}

```

In the above loop nests, array A was transformed from a 2-D array into a 1-D array. Similarly, array B was transformed from a 2-D array into a scalar variable.

Note that array A can be further contracted to a circular buffer of length 2 if we enable modular mapping in our implementation. This is left for future extension.

15 Polyhedral Scanning

After optimizations have been performed, the **R-Stream** mapper has to transform the mapped result in polyhedral form back to the form of the **R-Stream** internal representation, the Sprig IR. This code generation process is basically a form of loop synthesis, and is called *polyhedral scanning* in the polyhedral research community. This term arose from the geometric interpretation of the following operation: we are given a domain D , and we would like to generate a loop nest such that all integral points $x \in D$ are visited in lexicographical order. The above problem is the same as the problem of code generation for one statement. In the case of multiple statements, the problem generalizes to the following: we are given a set $D = \bigcup_{i=1 \dots n} D_i$, where D_i denote the iteration domain of statement S_i , and we would like to generate a set of loop nests which visits all the points $x \in D$ in the lexicographical order and execute S_i whenever $x \in D_i$.

Because of the complex transformations being performed in the **R-Stream** mapper, the output loop nests usually have very little similarity with the original. Thus the polyhedral scanning problem is not based on syntactic transformations, but rather on polyhedral operations.

15.1 Example

To illustrate the process of polyhedral scanning, let us begin with a simple example, taken from [BW94].

Suppose we are given the following one statement loop fragment to process in the mapper:

```
for (i = 10; i <= 15; i++)
  for (j = 1; j <= 3; j++)
    for (k = 1; k <= 50; k++)
      S(i, j, k);
```

The iteration space of the statement can be determined by the simple rectangular set:

$$\mathcal{D} = \{[i, j, k] \mid 10 \leq i \leq 15, 1 \leq j \leq 3, 1 \leq k \leq 50\}$$

Initially, the space-time mapping (or schedule) of the statement is the identity.

Let us suppose that the mapper has decided the following schedule Θ provides the best performance for the loop nest.

$$\Theta(i, j, k) = \begin{bmatrix} 0 & 6 & 1 \\ 1 & -3 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

We would like to regenerate a loop nest which implements this schedule. We can approach the problem as follows. Construct a new polyhedron T , defined as:

$$T = \{[t_1, t_2, t_3, i, j, k] \mid [t_1, t_2, t_3] = \Theta(i, j, k), (i, j, k) \in \mathcal{D}\}$$

A loop nest can be obtained by visiting the all the integral points in T in lexicographical order.

Since in this particular case Θ is unimodular, we can invert Θ in T and obtain:

$$\begin{aligned}\mathcal{D}' &= \{\Theta^{-1}(i, j, k) \mid [i, j, k] \in \mathcal{D}\} \\ &= \{[i, j, k] \mid 1 \leq i - 6k \leq 50, 10 \leq j + 3k \leq 15, 1 \leq k \leq 3\}\end{aligned}$$

By performing successive projections on \mathcal{D}' , we obtain the following sets:

$$\begin{aligned}\mathcal{D}_1 &= \mathcal{D} \\ &= \{[i, j, k] \mid 1 \leq k \leq 3, 1 \leq i - 6k \leq 50, 10 \leq j + 3k \leq 15\} \\ \mathcal{D}_2 &= \{[i, j] \mid 7 \leq i \leq 68, 1 \leq j \leq 12, 21 \leq i + 2j \leq 80\} \\ \mathcal{D}_3 &= \{i \mid 7 \leq i \leq 68\}\end{aligned}$$

By converting the constraints in these projections as upper- and lowerbounds on loop nests, we can obtain a three level nested loop as our final result:

```
for (i = 7; i <= 68; i++)
  for (j = max(⌈(21-i)/2⌉, 1);
       j <= min(12, ⌊(80-i)/2⌋); j++)
    for (k = max(1, ⌈(i-50)/6⌉, ⌈(10-j)/3⌉);
         k <= min(3, ⌊(i-1)/6⌋, ⌊(15-j)/3⌋); k++)
      S1(j+3k, k, i-6k);
```

15.2 Formal statement

The formal statement of the polyhedral scanning problem is as follows: given statements S_1, \dots, S_n with (parametric) iteration domains $D_1(y), \dots, D_n(y)$, and space-time (or schedules) mappings $\Theta_1, \dots, \Theta_n$, compute a set of loop nests which executes S_1, \dots, S_n in its given order.

Note that it is possible to combine the domains D_i with the schedules Θ_i in a combined encoding. For instance, define:

$$D'_i(y) = \{[z, x] \mid z = \Theta_i(x, y), x \in D_i(y)\}$$

Thus, applying polyhedral scanning on the sets D'_i is equivalent to scanning (D_i, Θ_i) .

A naive solution to the general polyhedral problem is to emit the following loop nests as a starting point and progressively simplify and optimize the resulting code:

```
for [z, x] ∈ ⋃i D'_i(y) in lexicographical order
  if  $\Theta_1(x) \in D_1(y)$  then S_1(x)
  ...
  if  $\Theta_n(x) \in D_n(y)$  then S_n(x)
```


There are however some difficulties to this approach. Among them are the following.

- The set $\bigcup_i D'_i(y)$ is in general non-convex, and thus cannot be easily rendered into a perfectly nested loop nests.
- The predicates within the inner loops are obstacles to performance.
- Heuristics to optimize the predicates and loop nests may not work in general. This is particularly true if local heuristics are employed.

15.3 Related works

There is a wealth of results in the past two decades on the polyhedral scanning problem [AI91, KPR95, KPR98, GLW98, QRW00, QR, Bas03, Bas04a]. Ancourt and Irigoin were one of the first researchers to study the polyhedral scanning problem restricted to a single loop statement and only unimodular transformations. Subsequent improvements extends the scope of the problem to non-unimodular transformations, and to multiple statements. The first algorithm to adequately resolve the general polyhedral scanning problem for multiple statements is the work of Quill  re and Rajopadye [QRW00], with subsequent performance and code quality improvements by Bastoul [Bas03, Bas04a]. Recent results by Vasilache build on top of those approaches and focus even more on code generation quality and efficiency [VBC06, Vas07].

15.4 R-Stream’s polyhedral scanner

R-Stream’s polyhedral scanner is based on the state-of-the-art algorithms of Quill  re et al. [QRW00] and Bastoul [Bas04a] as implemented in the CLooG polyhedral scanner [Bas04b]. Unlike the naive approach described earlier, these algorithms are based on recursive decomposition.

The basic algorithm of [QRW00] depends on two basic transformations:

1. Given one domain D , we can generate one perfect scanning loop nest by repeatedly performing projections on D until all the dimensions have been projected. This is the same process illustrated earlier in Section 15.1.
2. Given a set of potentially overlapping domains D_1, \dots, D_n , where $n > 1$, we can scan them as follows:

separation Separate them into a set of disjoint convex domains D'_1, \dots, D'_m .

polyhedral sorting Topologically sort D'_1, \dots, D'_m in a given dimension.

This is possible because the D'_1, \dots, D'_m are disjoint.

recursion Recursively scan the domains D'_1, \dots, D'_m .

The algorithm in [Bas04a] also contains additional transformations for removing redundant operations; for simplicity, we shall omit such details.

Figure 7 summarizes the Quill  re algorithm in pseudo-code.

```

quillere( $\mathcal{C}, \mathcal{T}, d$ )
  (1) Intersects the domains in  $\mathcal{T}$  with  $\mathcal{C}$ 
  (2) Project the domains in  $\mathcal{T}$ :
    for  $(D \rightarrow T) \in \mathcal{T}$  do
      Replace  $(D \rightarrow T)$  by  $proj_d(D) \rightarrow (D \rightarrow T)$ 
    end for;
  (3) Separate  $\mathcal{T}$  into trees with disjoint domains
  (4) Sort the separated  $\mathcal{T}$ 
  (5) for  $(D \rightarrow T) \in \mathcal{T}$  do
    quillere( $D, T, d + 1$ );
  end for;
  (5') Optionally apply the separation/sort steps
    (See Quillere's paper)
  (6) Simplify the domains of  $\mathcal{T}$  against  $\mathcal{C}$ 
  return  $\mathcal{T}$ ;

Separate( $\mathcal{P}$ )
   $\mathcal{Q} := \emptyset$ ;
  for  $P_i \in \mathcal{P}$  do
     $\mathcal{Q}' := \emptyset$ ;
    for  $Q_j \in \mathcal{Q}$  do
       $\mathcal{Q}' := \mathcal{Q}' \cup (Q_j - P_i)$  (with stms. of  $Q_j$ );
       $\mathcal{Q}' := \mathcal{Q}' \cup (Q_j \cap P_i)$  (with stms. of  $Q_j$  and  $P_i$ );
    end for;
     $\mathcal{Q}' := \mathcal{Q}' \cup (P_i - \cup_j Q_j)$  (with stms. of  $P_i$ );
     $\mathcal{Q} := \mathcal{Q}'$ ;
  end for;
  return  $\mathcal{Q}$ ;

```

Figure 7: Quillère's algorithm.

Our polyhedral scanner (called Bungle) is a reimplementation of CLooG [QRW00, Bas04b, Bas04a] in Java. The **R-Stream** mapper originally depended on CLooG for polyhedral scanning. However, we have discovered a few deficiencies in the algorithm of CLooG, related to output code quality. The performance improvements are a welcomed product of the reimplementation.

15.5 Performance improvements

Among the performance improvements are these changes to the Quill re algorithm:²¹

- We used data structure sharing whenever possible. With garbage collection in Java, this is trivial to accomplish; there is no need to perform reference counting as in CLooG.
- As in CLooG, we identify β -dimensions, i.e., dimensions in the domains which are constants [BCG⁺03a]. β -dimensions can be sorted using integer sort instead of polyhedra topological sort.
- Statements which have the same domains and adjacent schedules can merge into a single “block.” The polyhedral scanner can treat these blocks as a single statement.
- The separation algorithm has been tuned. Avoid difference, union and intersection whenever possible. Use cheaper \subseteq tests to prune out the common cases. Figure 8 shows the pseudo code of our fine-tuned separation algorithm.
- The polyhedral topological sorting algorithm has been sped up, using only one call to the Chernivoka algorithm [LeV92] per polyhedra comparison step, as opposed to six as described in [QRW00]. The Chernikova algorithm [Che68] is the primitive operation for performing Minkowski decomposition of polyhedra, and is the primitive on which all polyhedral set operations are built.
- Faster preimage and image primitives are used for the common cases of adding a new dimension to a set. These special cases can be computed without calling the relatively expensive Chernikova algorithm.

Additionally, we are currently working on other optimizations to improve the code generation speed even more:

- Perform aggressive node splitting after sorting at each level of the algorithm based on partial domain equality. This technique described in previous work by Vasilache [VBC06, Vas07] greatly reduces the number of redundant polyhedral separations.

²¹Many of these changes are taken from CLooG.

```

Separate( $\mathcal{P}$ )
 $\mathcal{Q} := \{P_1\}; Q_{union} := \{P_1\};$ 
for  $P_i \in P_2, \dots, P_n$  do
   $P\_in\_Qs := \text{false};$  // Is  $P_i \subseteq \mathcal{Q}$ ?
   $\mathcal{Q}' := \emptyset;$ 
  for  $Q_j \in \mathcal{Q}$  do
     $P\_in\_Q := (P_i \subseteq Q_j);$ 
     $Q\_in\_P := (Q_j \subseteq P_i);$ 
    // Process  $P_i \cap Q_j$ 
    if  $P\_in\_Q$  then  $P\_in\_Qs := \text{true}; PQ := P_i;$ 
    else if  $Q\_in\_P$  then  $PQ := Q_j;$ 
    else  $PQ := P_i \cap Q_j;$ 
    end if;
    if  $PQ \neq \emptyset$  then
       $\mathcal{Q}' := \mathcal{Q}' \cup PQ$  (with stms. of  $Q_j$  and  $P_i$ );
    end if;
    // Process  $Q_j - P_i$ 
    if  $Q\_in\_P$  then
      else if  $PQ \neq \emptyset$  then
         $\mathcal{Q}' := \mathcal{Q}' \cup Q_j$  (with stms. of  $Q_j$ );
      else if  $Q_j \not\subseteq PQ$  then
         $\mathcal{Q}' := \mathcal{Q}' \cup (Q_j - P_i)$  (with stms. of  $Q_j$ )
      end if;
    end for;
    // Process  $P_i - \cup_j Q_j$ 
    if not  $P\_in\_Qs$  and (not  $\text{convex}(Q_{union})$  or not  $P_i \subseteq Q_{union}$ ) then
       $\mathcal{Q}' := \mathcal{Q}' \cup (P_i - Q_{union})$  (with stms. of  $P_i$ );
      if  $P_i - Q_{union} \neq \emptyset$  then  $Q_{union} := Q_{union} \cup P_i;$  end if;
    end if;
   $\mathcal{Q} := \mathcal{Q}';$ 
end for;
return  $\mathcal{Q};$ 

```

Figure 8: Improved separation algorithm.

- Perform polyhedral topological sorting by only lexicographically comparing well-chosen vertices. This allows us to skip all Minkowski decomposition of the polyhedra.

Figure 2 shows the speedup of our polyhedral scanner over CLooG, running a benchmark suite provided by the CLooG distribution.

Platform	CLooG	Bungle	Speedup
Windows-XP/x86	108.66s	12.1s	9.0
Linux/x86	42.2s	9s	4.7
Linux/x86-64	25.67s	4.9s	5.2

Table 2: CLooG versus Bungle (64-bits).

15.6 Code quality improvements

As mentioned above, one of the main motivations for developing our polyhedral scanner was that we find the output code quality of CLooG inadequate. Two areas are particularly troublesome: Firstly, tiling transformations frequently produces strided loops, which CLooG generates as guarded code. Secondly, CLooG does not contain an adequate mechanisms to control code duplication. Note that there is a constant tradeoff between code duplication and specialization.

To attack these problems, we have included the following set of code quality improvements into Bungle:

- Perform less domain splitting/specialization.
- Perform constraints tightening to improve the bounds of the loop nests.
- Perform predicate hoisting and stride hoisting to improve the generation of predicated and strided code.
- Provide a mechanism to control when specialization should be performed (i.e., introduce code duplication), and when guards should be used (i.e., eliminate code duplication.)

We shall show how we implement these changes and how they improvement the output code quality in the next sections.

We are also working towards integrating recent results that translate into additional code quality improvements [VBC06, Vas07] such as:

- Taking advantage of the notion of schedule equivalence to perform Hermite Normal Form simplification of the scheduling to drastically reduce the amount of unnecessary modulo conditionals.

- Use unrolling to remove remaining modulo guards.
- Tightly integrate modulo conditionals with domain computations to remove dead code that is not found as such by subsequent low level optimizations.
- Exploit new properties of schedule equivalence to avoid performing separations in the Quilleré algorithm at certain strategic points.

15.6.1 Controlling domain splitting

To control excessive domain splitting (code duplication), we have disabled the bottom-up recursion in the Bastoul [Bas04a] algorithm. This recursion step intends to eliminate unnecessary control overhead in the generated loop. But unfortunately, it tends to introduce extra code specialization.

For example, suppose we start with the following loop nests and apply a fusion transformation to improve its spatial locality.

```
if (M >= 0) {
  for (int i = 0; i <= N; i++) {
    for (int j = 0; j <= M; j++) {
      A[i][j] = f(C[-2 + i][1 + j]);
    }
    for (int j = 0; j <= M; j++) {
      B[i][j] = g(A[i][1 + j], A[i][j], C[-1 + i][j]);
    }
    for (int j = 0; j <= M; j++) {
      C[i][j] = h(B[i][j], A[i][2 + j], A[i][1 + j]);
    }
  }
}
```

One possible result after fusion is the following loop nests:

```
for (int i = 0; i <= N; i++) {
  for (int j = -2; j <= M; j++) {
    if (j <= M - 2) {
      A[i][2 + j] = C[-2 + i][3 + j];
    }
    if (j >= 0) {
      B[i][j] = A[i][1 + j] + A[i][j] + C[-1 + i][j];
      C[i][j] = A[i][2 + j] + A[i][1 + j] + B[i][j];
    }
  }
}
```

However, the above loop nests are clearly inefficient, because of the presence of predicates in the innermost loop. CLooG attempts to remove these predicates by splitting domains in the separation step of the algorithm. The result is the following:

```

if (M >= 2)
  for (int i = 0; i <= N; i++) {
    for (int j = -2; j <= -1; j++)
      A[i][2 + j] = C[-2 + i][3 + j];
    for (int j = 0; j <= M + -2; j++) {
      A[i][2 + j] = C[-2 + i][3 + j];
      B[i][j] = A[i][1 + j] + A[i][j] + C[-1 + i][j];
      C[i][j] = A[i][2 + j] + A[i][1 + j] + B[i][j];
    }
    for (int j = M + -1; j <= M; j++) {
      B[i][j] = A[i][1 + j] + A[i][j] + C[-1 + i][j];
      C[i][j] = A[i][2 + j] + A[i][1 + j] + B[i][j];
    }
  }
if (M <= 1 && M >= 0)
  for (int i = 0; i <= N; i++) {
    for (int j = -2; j <= M + -2; j++)
      A[i][2 + j] = C[-2 + i][3 + j];
    for (int j = 0; j <= M; j++) {
      B[i][j] = A[i][1 + j] + A[i][j] + C[-1 + i][j];
      C[i][j] = A[i][2 + j] + A[i][1 + j] + B[i][j];
    }
  }

```

The bottom-up recursion in CLooG has removed unnecessary code in the special of $0 \leq M \leq 1$. However, it is accomplished by code duplication. In Bungle, we defaulted to the original Quill re algorithm (which does not apply bottom-up recursion). The following simpler output is generated:

```

if (M >= 0) {
  for (int i = 0; i <= N; i++) {
    for (int j = -2; j <= min(M + -2, -1); j++) {
      A[i][2 + j] = C[-2 + i][3 + j];
    }
    for (int j = 0; j <= M + -2; j++) {
      A[i][2 + j] = C[-2 + i][3 + j];
      B[i][j] = A[i][1 + j] + A[i][j] + C[-1 + i][j];
      C[i][j] = A[i][2 + j] + A[i][1 + j] + B[i][j];
    }
    for (int j = max(0, M + -1); j <= M; j++) {
      B[i][j] = A[i][1 + j] + A[i][j] + C[-1 + i][j];
    }
  }
}

```

```

        C[i][j] = A[i][2 + j] + A[i][1 + j] + B[i][j];
    }
}

```

15.6.2 Constraints tightening

We have found that constraints tightening has a significant effect on the output quality of polyhedral scanning after tiling transformations are applied. For example,

```

for (int i = 0; i <= 8; i++)
  for (int j = 0; j <= 8; j++)
    for (int k = 0; k <= 8; k++)
      for (int l = max(2*i, 0); l <= min(64, 2*i+63); l++)
        for (int m = max(2*j, 0); m <= min(64, 2*j+63); m++)
          for (int n = max(2*k, 0); n <= min(64, 2*k+63); n++)
            S;

```

All of the `min` and `max` operations are redundant, and if left behind in the code, will have drastic negative effect on LLC optimizations.

We have observed that by tightening the constraints, we can obtain the following redundancy-free loops:

```

for (int i = 0; i <= 8; i++)
  for (int j = 0; j <= 8; j++)
    for (int k = 0; k <= 8; k++)
      for (int l = 2*i; l <= 2*i+63; l++)
        for (int m = 2*j; m <= 2*j+63; m++)
          for (int n = 2*k; n <= 2*k+63; n++)
            S;

```

Our constraints tightening heuristics is very simple: after every projection operation in the Quill re algorithm, we tighten the constraints of the polyhedra by performing two types of Gomory cuts:

- Given a constraints $ax + b \geq 0$, we can replace it by $a/gx + \lfloor b \rfloor / g \geq 0$, where g is the gcd of the coefficients in a .
- We compute the minimum and maximum value of each dimension of a polyhedron (if they exist) by performing a projection in each dimension, and add these minimum and maximum as extra constraints.

15.6.3 Predicate and stride hoisting

Modulo equality predicates within a loop can often be transformed by hoisting them and converting them into strided loops. For instance, consider the following output generated by CLoog:

```
for (int i = 1; i <= min(N,6); i++) {
  for (int j = 1; j <= min(N,(-(i) + 7)); j++) {
    if (((i + 1) & 1) == 0) {
      S1(i,j,((-1 + i) / 2),N);
      S2(i,j,((-1 + i) / 2),N);
    }
  }
  for (int j = (-(i) + 8); j <= N; j++) {
    if (((i + 1) & 1) == 0) {
      S1(i,j,((-1 + i) / 2),N);
    }
  }
  for (int j = (N + 1); j <= (-(i) + 7); j++) {
    if (((i + 1) & 1) == 0) {
      S2(i,j,((-1 + i) / 2),N);
    }
  }
}
```

Unfortunately, the predicates within the innermost loops are serious hindrance to performance.

Since all of the predicates are loop invariant, Bungle will actually hoist them out of their respective loops and combine them into a single predicate:

```
for (int i = 1; i <= min(N,6); i++) {
  if (((i + 1) & 1) == 0) {
    for (int j = 1; j <= min(N,(-(i) + 7)); j++) {
      S1(i,j,((-1 + i) / 2),N);
      S2(i,j,((-1 + i) / 2),N);
    }
    for (int j = (-(i) + 8); j <= N; j++) {
      S1(i,j,((-1 + i) / 2),N);
    }
    for (int j = (N + 1); j <= (-(i) + 7); j++) {
      S2(i,j,((-1 + i) / 2),N);
    }
  }
}
```

Finally, the remaining predicate can be combined with the i-loop, by modifying the stride of the loop to 2. The final result is predicate free:

```

for (int i = 1; i <= min(N,6); i += 2) {
  for (int j = 1; j <= min(N,(-(i) + 7)); j++) {
    S1(i,j,((-1 + i) / 2),N);
    S2(i,j,((-1 + i) / 2),N);
  }
  for (int j = (-(i) + 8); j <= N; j++) {
    S1(i,j,((-1 + i) / 2),N);
  }
  for (int j = (N + 1); j <= (-(i) + 7); j++) {
    S2(i,j,((-1 + i) / 2),N);
  }
}

```

Note that in general, the hoisting and stride combination process is more intricate for more general predicates. Consider the following sequence of transformations:

```

for (int i = 0; i <= 30; i++) {
  for (int j = 0; j <= 30; j++) {
    if (((((i + (3 * j)) + 1) % 6) == 0)) {
      S1(i,j,((( -1 + (-1 * i)) + (3 * j)) / 6));
    }
  }
}

```

```

for (int i = 0; i <= 30; i++) {
  if (((i + 1) % 3) == 0) {
    gap1 = (((-i+5)/3 & 1);
    for (int j = gap1; j <= 30; j += 2) {
      S1(i,j,((( -1 + (-1 * i)) + (3 * j)) / 6));
    }
  }
}

```

```

for (int i = 2; i <= 30; i += 2) {
  gap1 = (((-i+5)/3 & 1);
  for (int j = gap1; j <= 30; j += 2) {
    S1(i,j,((( -1 + (-1 * i)) + (3 * j)) / 6));
  }
}

```

General transformation rule in stride hoisting is as follows. Given a loop nest matching the following template:

```

for (i = l; i <= h; i++)
    if ((e + ai) % m) == 0)
        S;

```

where $pa + qm = \gcd(a, m) = g$, we can transform the loop nest into:

```

if (e % g == 0) {
    gap = ((m - p(e + al))/g) rem (m/g);
    for (i = l + gap; i <= h; i += m/g)
        S;
}

```

15.6.4 Controlling code duplication

The final improvement to Bungle is the ability to tradeoff specialization and code duplication. An example will illustrate the importance of this feature. Consider the following loop nests generated by CLooG for a block matrix multiply kernel using double buffering to overlap communication and computation.

```

for (i = 0; i <= 7; i++) {
    dma get A, B, C
    dma wait
    swap pointers to A, B, and C
    dma get A, B, C
    for (j = 16 * P; j <= 16 * P + 15; j++)
        for (k = 0; k <= 15; k++)
            for (l = 16 * i; l <= 16 * i + 15; l++)
                C[j-16*P][k] += B[l-16*i][k] * A_1[j-16*P][l-16*i];
    dma put C
    for (j = 1; j <= 6; j++) {
        dma wait
        swap pointers to A, B, and C
        dma get A, B, C
        for (k = 16*P; k <= 16*P + 15; k++)
            for (l = 16 * j; l <= 16 * j + 15; l++)
                for (m = 16 * i; m <= 16 * i + 15; m++)
                    C[k-16*P][l-16*j] += B[m-16*i][l-16*j] *
                        A[k-16*P][m-16*i];

        dma wait
        dma put C
    }
    dma wait
    swap pointers to A, B, and C
    for (j = 16 * P; j <= 16*P + 15; j++)
        for (k = 112; k <= 127; k++)

```

```

        for (l = 16 * i; l <= 16 * i + 15; l++)
            C[j-16*P][k-112] += B[l-16*i][k-112] * A[j-16*P][l-16*i];
        dma wait
        dma put C
        dma wait
    }

```

Note that the main loop kernel has been specialized 3 times and multiple copies of DMA operations have been inserted. This code duplication behavior is fundamental to the Quillère algorithm, as it tries to remove unnecessary predicates in the separation step of the algorithm. Uncontrolled specialization, however, can blow up the size of the transformed loop substantially.

By removing specialization outside of the main computation kernel, Bungle can produce the following simpler loop nest:

```

for (i = 0; i <= 7; i++) {
    for (j = -1; j <= 8; j++) {
        if (j <= 7 && j >= 0) {
            dma wait
            swap pointers to A, B, C
        }
        if (j <= 6) {
            dma get A
            dma get B
            dma get C
        }
        if (j <= 7 && j >= 0)
            for (k = 16*P; k <= 16*P + 15; k++)
                for (l = 16*j; l <= 16*j + 15; l++)
                    for (m = 16*i; m <= 16*i + 15; m++)
                        C[k-16*P][l-16*j] += B[m-16*i][l-16*j] *
                                                A[k-16*P][m-16*i];
        if (j >= 1) dma wait
        if (j <= 7 && j >= 0) dma send
    }
}

```

This set of loop nests is vastly superior in terms of code size. Since the predicates are only inserted outside of the innermost matrix matrix loop nests, their performance penalty is negligible.

The heuristics to achieve this improved output is as follows:

- The polyhedral scanner takes as input a range of loop nest dimensions to allow specialization for each input statement. Such ranges are identified in the tiling phase of the mapper: dimensions which belong within a tile are computationally intensive, and so specialization is allowed. Dimensions

which are outside of a tile (so called inter-tile dimensions) are not allowed to be specialized.

- The Quillère algorithm is modified so that when it encounters statements which are not allowed to be specialized, it generates the simpler non-specialized result.

16 The R-Stream Compiler Infrastructure

Figure 9 shows the structure of the entire R-Stream 3.0 source-to-source compiler, together with the mapper component. The core of the compiler consists of the front-end, the intermediate representation (IR) and scalar optimizer known as Sprig, and the back-end for regenerating the optimized code in a source-level format. The polyhedral mapper, and the components for translating between the Sprig IR form and the polyhedral form (called “raising” and “lowering”) are considered to be an extension of the compiler. Sections 17 and 18 will discuss these transformations in more detail.

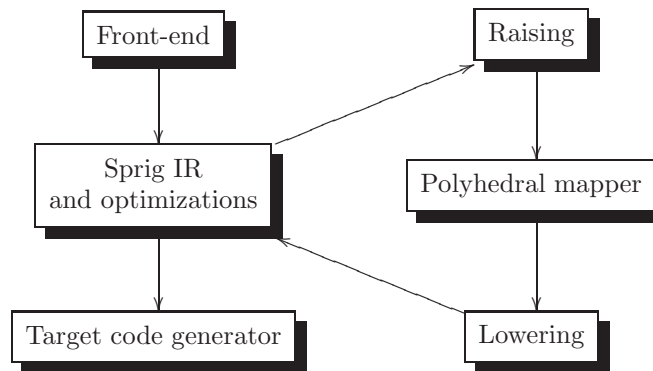


Figure 9: The R-Stream infrastructure.

16.1 The Sprig IR

The Sprig IR is a graph-based static single assignment form (SSA) intermediate representation inspired by the work of Click et al. [CP95], with some extensions borrowed from the value dependence graph (VDG) [WCES94] for representing state and memory. As in a traditional compiler, Sprig ignores many of the syntactic artifacts of the source language and concentrates only on the semantics. The Sprig IR is structured as a single unified fine-grained operator graph in which most dataflow and data dependencies are represented explicitly. At the lowest level, an operator denotes a primitive operation such as addition, subtraction, or a load and store from memory. Dependencies based on memory are explicitly represented as extra edges in memory operators. Control flow within a function or procedure is represented by a control flow graph (CFG) embedded on top of the operator graph data structure. At the highest level, procedural calls and returns are represented by call and return operators (`APP` and `RETURN`) connected to function operator (called `LAMBDA`s). Because the entire compilation unit forms a single unified graph, this makes Sprig very suitable for interprocedural and full program analysis.

16.1.1 Heterogeneous compilation

Heterogeneous compilation imposes additional requirements to the design of the Sprig IR. While a traditional compiler only has to manipulate one target architecture at a time, the R-Stream compiler frequently has to manipulate and combine target code in multiple architectures. For example, when producing a mapping for the CELL platform, we have to produce mapped control code running on PowerPC and mapped kernels running on CELL's Synergistic Processor Units (SPUs).

Thanks to judicious use of the object oriented design paradigm, the R-Stream compiler can manipulate multiple IR instances of different architectures at the same time. To handle heterogeneous architectures, each IR instance can be parameterized by a different architectural description object, which describes the target machines' parameters such as word size, address width, memory and cache sizes, etc.

16.1.2 Source level type system

One distinguishing feature of the Sprig IR is that all operators are typed using source level types (in this case types from the C language). This is a direct requirement of the source-to-source nature of the compiler.

Since the C type system is largely ad hoc and irregular, a few accommodations are needed to make it possible to work within a compiler intermediate representation.

First, all type coercions and conversions are explicitly represented in the IR. This means that unlike the C language in the source level, all type coercions and type conversions are expanded and explicitly represented in the IR. For example, the C fragment

```
void * p = ...  
long long x = (long long) p;
```

may be represented internally as the following two operators:

```
x <- SX.64.32(long long, CAST(int, p))
```

Here `CAST` is a type cast operator and `SX` sign-extension operator and the purpose of `SX.64.32` is to sign-extend a 32-bit integer into a 64-bit integer `long long`.

Secondly, all optimizations in Sprig function in a type preserving manner. Because all type conversions and coercions are explicitly represented, the internal type rules operate very similarly to that of the simply typed lambda calculus, rather than the complex rules adopted by the C language. Thus the extra maintenance required for typed transformation is minimal. As an extra sanity check, the R-Stream compiler can also perform a full type check after every optimization pass.

As a hedge for future extension to other source languages, the set of types and type rules are also fully parametrizable in the Sprig IR, rather than hardwired into the representation.

16.1.3 Source code regeneration

Another distinguishing feature of the Sprig IR infrastructure is the ability to regenerate the internal representation in source code form. This is needed because Sprig IR strips away all source level artifacts in the internal representation.

While this design is in contrast to many existing source-to-source program transformation tools – which tend to retain the source syntax in their internal representations – we believe it is the right one. By ignoring the source syntax and only concentrate of the semantics of the input program, we obtain two main advantages:

- Transformations and analyses are much easier to develop because they are not sensitive to syntactic restrictions or irregularities of the underlying language(s).
- Retargeting to another source and/or target language is easier, because the compiler is not tied to one specific source or target language.

Of course there are other potential disadvantages of this approach. The most important one is that the output code will tend not to resemble the input source syntactically in any way. However, as we will show in Section 18, many low level compilers (LLC) that accept **R-Stream** output require different idioms to work properly. So the chosen idiom in which the source is presented may not be what the LLC expects. Giving **R-Stream** the ability to customize the output syntax to fit the target compilers is both a necessity and an advantage.

Section 18 will cover the source code regeneration process in more detail.

16.2 Scalar optimizations and analyses

The Sprig IR infrastructure contains an extensive list of SSA-based scalar optimizations, including conditional constant propagation [WZ91], global value numbering and code motion [Cli95, CS95, Sim98], strength reduction [CSV01], global reassociation [BC94], dead code elimination, and function inlining.

All these optimizations and analyses are used to clean up and simplify the input code before mapping begins, and also to clean up and perform post-mapping transformations on the mapped code.

17 Raising: IR to Polyhedral Form

Raising is the process of transforming loop nests in the form of the the Sprig IR (see Section 16) into the internal polyhedral form (See Section 4) needed by the mapper. Note that raising is trivial when restricted to an input representation that matches the semantics restriction of static affine control programs exactly, i.e., do-loops with affine loop bounds and affine indexing expressions on arrays. However, when performed on realistic programs and intermediate representations, the process is more complicated.

First, there is a semantic mismatch between the polyhedral model and the internal representation of a typical compiler. For example, while the polyhedral model expects loop nests, a compiler is often given a control flow graph, and thus loop detection is required to transform one to the other. Similarly, while the polyhedral model expects arrays and array indexing, a compiler typically deals with lower level address arithmetic, with all the difficulties (such as aliasing) that entails.

Secondly, since the C language is a low level system programming language and is insufficiently expressive, programmers are frequently forced to encode many high level concepts — such as abstract data types, dynamically sized multidimensional arrays, etc. — as more primitive C constructs. The difficulty posed to the raising phase is that it has to perform “semantics raising” to recover the original intention of the programmer by decoding such encodings. Of course, the set of such possible idiomatic encodings is limitless and it is impossible for the raising phase to perform a “mind-reading” of the programmer in all the possible cases. Nevertheless, by examining sample applications in the target DSP and high performance computing area, we have identified a collection of common idioms that can be automatically decoded. Our raising algorithm has incorporated some of these analyses.

Finally, the pure polyhedral model is overly restrictive, and if interpreted strictly, will prohibit many existing loop nests from being handled by the mapper. For example, data dependent branches, function calls and returns, etc. are common “impure” constructs found in many program fragments. To be effective, the raising phase must be able to handle all these constructs gracefully.

17.1 Raising algorithm

The raising algorithm implemented in the **R-Stream** compiler contains the following subanalyses and transformations, each handling an aspect of the raising problem. Currently, these are pointer-analysis; mappable region identification; inlining; index, data and predicate identification; if-conversion; base address and parameters detection; statement formation; recurrence analysis; and finally, GDG building.

In the next sections we shall describe how all these components work together to form the raising phase.

17.1.1 Pointer analysis

The first step of the raising process is to perform a whole program pointer analysis to determine potential aliasing. This analysis computes a global object graph, which summarizes all the potential points-to relationships in the program. Currently, we use a flow-insensitive, context-insensitive, subset based pointer analysis based on the research by Heintze and Tardieu [HT01, WL02]. Our variant of this analysis is also field-sensitive, i.e., can distinguish between different members within a **struct** or **union**.

```
typedef struct {
    int n;
    float *A, *B, *C;
} obj;

void matmult(obj * p) {
    int n = p->n;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            p->C[i*n+j] = 0;
            for (k = 0; k < n; k++)
                p->C[i*n+j] += p->A[i*n+k]*p->B[k*n+j];
        }
    }
}

void f() {
    obj * o = (obj*)malloc(sizeof(obj));
    o->n = 1024;
    o->A = (float*)malloc(sizeof(float)*1024*1024);
    o->B = (float*)malloc(sizeof(float)*1024*1024);
    o->C = (float*)malloc(sizeof(float)*1024*1024);
    ...
    matmult(o);
    ...
    float * q = o->A;
    ...
}
```

Figure 10: Pointer analysis example

Figure 11 shows the result of applying this pointer analysis to the program fragment in Figure 10. Dotted edges are interpreted to be inclusion edges in the points-to relationship. For example, there is a dotted edge from **p** to **o**. Thus, from the pointer **p** we can reach all the objects in which **o** may point-to. Solid labeled edges are interpreted as **member** selection operations. Thus from the

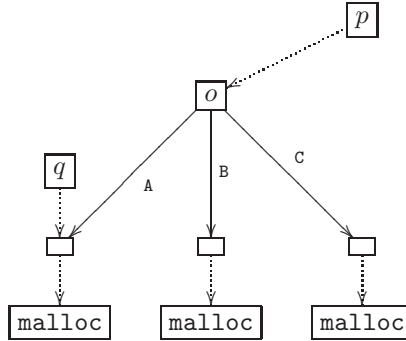


Figure 11: Object summary graph

graph we know that $p \rightarrow A$, $p \rightarrow B$ and $p \rightarrow C$ cannot alias because $o \rightarrow A$, $o \rightarrow B$ and $o \rightarrow C$ ultimately include distinct calls to `malloc`²². However q and $o \rightarrow A$ may alias, because they may reach the same `malloc` object in the graph.

17.1.2 Mappable region identification

After pointer aliasing, we determine which regions in the entire application can benefit from mapping. Currently, we defer this decision to the programmer, who annotates mappable functions with the pragma `#pragma rstream raise`, which identifies to the compiler that the annotated function may be raised. In the future we expect this step will be automated by static and dynamic profiling techniques.

To make the mappable region useful for subsequent transformation and analysis, we also perform an “outlining” transformation at this step, which has a secondary effect of exposing all the free variables that occur within the region. For example, given the following program fragment

```

#pragma rstream raise
void f() {
    for (int i = 0; i < N; i++) {
        A[i] = B[i] * C[i];
    }
}

```

outlining transforms this into the following equivalent fragment:

```

#pragma program rstream raise
void f() {
    _f(A, B, C);
}

```

²²We know that each `malloc` call returns a new object.

```

void _f(double *A, double *B, double *C, int N) {
    for (int i = 0; i < N; i++) {
        A[i] = B[i] * C[i];
    }
}

```

The freely occurring variables `A`, `B`, `C` and `N` are now rendered into parameters of the outlined function. These parameters are treated specially in the base address and parameters detection phase described in Section 17.1.7.

17.1.3 Inlining

To expose more potential parallelism to the mapper, small routines called from a mappable region may be inlined. Thus after mappable function identification, we run an inlining phase to perform this scope enlargement transformation. Currently, this phase depends on the `pragma rstream inline` inserted by the programmer to identify inlining opportunities. The rules are as follows.

- Functions with `#pragma rstream inline` annotated are inlinable.
- Function calls with `#pragma rstream inline` pragma are also inlinable.
- However, recursive functions are not inlined.

Similar to the mappable region identification phase, we expect this profitability computation to be automated in the future.

17.1.4 Index, data and predicate values classification

In the polyhedral model loop indices and data arrays are distinct entities. Since this is not directly evident in the SSA-based IR, after inlining we run a simple analysis to classify all SSA values that occur within a mappable region into three categories:

index values that are used directly or indirectly in loop bounds and array index computation,

data values that are used in arithmetic computation and loaded from and stored to memory, and

predicate values that control non-loop branches.

This analysis can be implemented as multiple traversals of the SSA graph. For example, to identify index values, we start from the addresses of all load and store operators, and visit and mark all reverse reachable operators.

Note that the above categories are not mutually exclusive; a value may in fact take the roles of all three. For example, consider the following source program fragment:

```

for (i = 0; i < 100; i++) {
    if (i % 3 == 0) {
        A[i] = i;
    }
}

```

The loop index `i` is used in array index computation, is stored inside an array and is also part of a predicate computation.

17.1.5 If-conversion

The polyhedral model does not handle data dependence predicates in its purest form. Since many existing programs contain data dependent predicates of some form, some compromise is necessary. To this end we use the standard technique of if-conversion [AKPW83], which transforms control dependences into data dependences.

Our if-conversion algorithm is a modification of existing algorithms and can work with arbitrary reducible flowgraphs. It is composed of these steps:

- Compute the control dependence graph (CDG) [FOW87] of the mapped region.
- Perform loop detection and identify back-edges.
- Within each nested loop, sequentialize all basic blocks with by topologically sorting of the forward control flow graph, i.e., by ignoring all the backedges.
- Convert each non-loop controlling predicate with into a predicate defining statement.
- Attach to each basic block a predicate expression computed from the forward control dependence graph, i.e., by ignoring all the back-edges. Predicate expressions are represented internally in our implementation as binary decision diagrams [Bry86], so boolean simplification is performed automatically.

The output of this step is a **predicated statement tree** consisting of loop nests and predicated “statements” that are at the granularity of basic blocks.

For example, suppose we are given the following program fragment:

```

void inc(double A[N][N]) {
    int i, j;
    for (i = 0; i < N-1; i++) {
        if (A[i][i] > 0) { // S1
            for (j = 0; j < N; j++) {
                if (A[i][j] > 0) { // S2

```

```

        A[i][j] += 1;    // S3
    } else {
        A[i+1][j] -= 1; // S4
    }
}
}
}
}
}
}

```

The if-conversion phase sinks all predicates into the innermost loops and converts all statements into predicated form, resulting in the a predicated statement tree that can be represented as the following pseudo code:

```

#pragma rstream map
void inc(double A[N][N]) {
    int i, j;
    for (i = 0; i < N-1; i++) {
        p1 = A[i][i] > 0; // S1
        for (j = 0; j < N; j++) {
            p2 = A[i][j] > 0 if (p1); // S2
            A[i][j] += 1 if (p1 && p2); // S3
            A[i+1][j] -= 1 if (p1 && !p2); // S4
        }
    }
}
}

```

With this transformation, control dependencies on the data dependent predicates in S1 and S2 are converted into data dependences on the predicate variables p1 and p2, and can then be treated as such in the mapper. Note that our mapper’s dependence analysis is predicate-aware, and thus will ignore all dependencies between statements S3 and S4 within the same iteration, since the boolean expression `p1 && p2` and `p1 && !p2` cannot be satisfiable at the same time.

17.1.6 Statement formation

The Sprig IR is structured as an operator graph where each operator is at the granularity of individual arithmetic or memory operation. While it is possible to map each operator into a node in the GDG, the resulting GDG is often too fine-grained. On the other hand, treating each basic block as a single unit often errs in the opposite extreme, and the resulting GDG is too coarse-grained, and may not contain enough exploitable parallelism. Thus to build “polyhedral statements” that have the right granularity for mapping, we run a special statement formation phase. This phase splits each basic block into multiple

polyhedral statements, each corresponding to a node in the GDG, using following heuristics:

- For each basic block, we build a data dependence graph summarizing all the dependencies involving non-index values. We ignore dependencies involving index values because they can be recomputed from the loop indices.
- We divide the basic blocks into partitions, such that all the partitions can be executed in some linear order. Partitions are built by repeating the follow process: we start from singleton nodes, and repeated merging in true-dependence edges as long as the linear order restriction is not violated.

The basic intuition is that only true dependence on non-index values require data communication, and partitions with no or with only false dependences can often be scheduled independently. Our heuristics thus seek to split basic blocks into the largest partitions without data communication. The result of this step is a modified predicated statement tree, consisting of loops and predicated statements, where each statement is a group of operators in the original graph.

17.1.7 Base address and parameters detection

The purpose of base address and parameters detection step is to identify which values are **base addresses** of variables and which values are **system parameters**, i.e., symbolic integer constants that can appear in indexing expressions and loop bounds. These information are used subsequently in recurrence analysis.

Currently, we employ the following heuristics:

- Pointer formal parameters that appear in the outlined functions of mappable regions (see Section 17.1.2) are tentative assumed to be a base address, and
- Integer formal parameters from the same functions are tentative assumed to be integer parameters.

17.1.8 Recurrence analysis

The second to last step of the raising phase is a recurrence (or induction) analysis, which perform these tasks:

- Identifies all do-loops within the program.
- Assigns a pseudo loop counter to all do-loops. Each loop counter is implicitly initialized to 0 at loop entry, and is incremented by one whenever the loop is repeated.
- Detects and identifies all recurrences involving integer and pointers within the program.

The effect of this phase is to reexpress an index value as a function of the loop indices and system parameters, and to reexpress an address computation as a function of the loop indices, system parameters and special base addresses computed in the base address detection phase.

For this purpose we utilize the algorithm of Pop et al. [PCS05], which allows us to detect linear recurrences, and other forms of progressions.

For example, consider the following source program fragment consisting of a single while loop.

```
void f1(int* A) {
    int i = 1;
    int j = 3;
    int n = 123;
    while (1) {
        int t;
        if (i >= n) break;
        t = i + 7;
        j = j + t;
        A[j] = i;
        i = i + 5;
    }
}
```

In the internal SSA form, the program is:

```
void f1(int* A) {
    int n = 123;
    while (1) {
        i1 =  $\phi(1, i_2)$ ;
        j1 =  $\phi(3, j_2)$ ;
        if (i1 >= n) break;
        t = i1 + 7;
        j2 = j1 + t;
        A[j2] = i1;
        i2 = i1 + 5;
    }
}
```

Recurrences can be detected by examining cycles that involve ϕ nodes in loop headers within an SSA graph. In this particular example, we can extract two cycles, one involving the *i* variable and one involving *j*:

```
i1 =  $\phi(1, i_1 + 5)$ 
j1 =  $\phi(3, j_1 + i_1 + 7)$ 
```


Translating the cycles into mathematical terms, we obtain the following sums:

$$\begin{aligned}
i &= 1 + \sum_{p=0}^{k-1} 5 \\
j &= 3 + \sum_{p=0}^{k-1} 7 + i \\
&= 3 + \sum_{p=0}^{k-1} 7 + 1 + \sum_{q=0}^{k-1} 5
\end{aligned}$$

where k is a newly introduced normalized loop index.

Using Newton's formula, we can collapse the two sums into the following closed forms:

$$\begin{aligned}
i &= 1 + 5k \\
j &= 3 + \frac{11}{2}k + \frac{5}{2}k^2
\end{aligned}$$

From the closed form of i and the loop limit n , 123, we know the while-loop is executed 25 times. Rewriting all loop indices in terms of a new 0-based loop index k , we obtain the following normalized loop:

```

void f1(int* A) {
    for (int k = 0; k < 25; k++) {
        i = 1+5k;
        j = 3+11/2*k+5/2*k*k
        A[11+21/2*k+5/2*k*k] = i;
    }
}

```

As stated, our algorithm also performs recurrence analysis on pointers and pointer arithmetic expressions. The generalization of this from integer expressions is immediate: each pointer expression closed form is an pointer recurrence expression of the form

$$A[e_1][\dots][e_n]$$

where A is a some recognized base-address (see Section 17.1.7) to an array and e_1, \dots, e_n are integer recurrence expressions. The analysis of recurrence expressions e_1, \dots, e_n are the same as outlined before.

By performing recurrence analysis on pointer expressions, we can automatically normalize programs written in C's pointer-centric style into an array expression required by the polyhedral model. Thus the programmer is not constrained by the needs of our mapper to rewrite his programs in another way.

For example, consider the following program fragment taken from [FO01]:

```

#define X 256
#define Y 256
#define Z 256
int A[X*Y];
int B[X*Y];
int C[X*Y];
void f() {
    int * p_a = &A[0];
    int * p_b = &B[0];
    int * p_c = &C[0];
    int k, i, f;
    for (k = 0; k < Z; k++) {
        p_a = &A[0];
        for (i = 0; i < X; i++) {
            p_b = &B[k*Y];
            *p_c = *p_a++ * *p_b++;
            for (f = 0; f < Y-2; f++) {
                *p_c += *p_a++ * *p_b++;
            }
            *p_c++ += *p_a++ * *p_b++;
        }
    }
}

```

This program fragment uses three scanning pointers `p_a`, `p_b` and `p_c` within the loops to traverse the arrays `A` and `B` and `C`.

By performing pointer recurrence analysis, we can normalize the above loop nests into the following array-centric form:

```

for (i = 0; i <= 255; i++) {
    for (j = 0; j <= 255; j++) {
        C[256*i+j] = A[256*i] * B[256*j];
        for (int k = 0; k <= 253; k++) {
            C[256*i+j] += A[1+256*j+k] * B[1+256*i+k];
        }
    }
}

```

17.1.9 GDG building

After all the previous raising steps, the final GDG building step traverses the predicated statement tree and translates all the gathered information into the GDG form. This phase converts all loops bounds into constraints, all indexing expressions into access functions, all predicate computation expressions and predicated statements into nodes of the GDG.

17.2 Future extensions

We shall conclude this section by describing some of the extensions we intend to add to the raising phase in the near future to make it handle a wider class of programs.

17.2.1 Automatic region selection and inlining

One current deficiency of the raising phase is that the R-Stream compiler cannot automatically determine which are the proper regions in the application to map. Currently, programmers have to insert `#pragmas` to direct this selection process. We intend to alleviate this lack with a combination of automatic and user-directed techniques:

- Perform a static program profile analysis [BL93, WL94] to determine the “hot spots” of an application. Then using the hot spots as a guide to select the mapping regions.
- Perform inlining of frequently used loop nests within the mapping regions to enlarge the scope of optimization.
- Allow user directed mapping region selection and inlining hints.

17.2.2 Abstract data types via struct and unions arguments

Proper programming practice states that one should encapsulate related data into abstract data types. Since the C language lacks such abstraction facilities, a common substitute is to package up related data into `structs` or `unions`, and implement functions that operate on such packaged data.

For example, Figure 10 shows such an example where three $n \times n$ matrices are packaged up into a single `struct` to be passed to a matrix multiply routine. Note that such practice is prevalent in libraries such as [SJHM06], or systems such as Matlab.

To analyze such programs, we intend to use the result of our pointer analysis to untangle the web of points-to relationships between C `structs` and pointers. In particular, in the above example, we have to infer the pointer expressions `p->A` and `p->B` and `p->C` are to be treated as base references to three separate arrays.²³

17.2.3 Heap memory management and array delinearization

Related to the above issue of abstract data type is the use of heap allocation. For example, one possible way to encode a 3-dimensional dynamically sized array with dimensions `m`, `n` and `o` is to encode it as a 1-D array allocated from the heap.

²³Note that this is not as trivial as it sounds, because in the C language pointer expressions are prevalent.

```

int m = ...
int n = ...
int o = ...
double * A = (double*)malloc(m*n*o*sizeof(double));
...
free(A);

```

A typical use of such “flattened” arrays is

```

int m = ...
int n = ...
int o = ...
double * A = (double*)malloc(m*n*o*sizeof(double));
double * p = A;
for (i = 0; i < m; i++, p += n*o)
    for (j = 0; j < n; j++, p += o)
        for (k = 0; k < o; k++)
            *p++ = ...

```

By performing pointer recurrence analysis, we can derive the following loop nests with array indexing rather than pointer scanning.

```

int m = ...
int n = ...
int o = ...
double * A = (double*)malloc(m*n*o*sizeof(double));
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < o; k++)
            A[i*n*o+j*o+k] = ...

```

Unfortunately, our polyhedral mapper cannot handle the non-affine indexing expressions $i*n*o+j*o+k$ directly. Fortunately, in this case we have a simple cure. The trick is to recognize that the original intention of the programmer is to encode a 3-D array as a flattened 1-D structure. Thus by *delinearizing* array A, we can obtain the following 3-D array indexing loop.

```

for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < o; k++)
            A[i][j][k] = ...

```

The above process requires some amount of pattern matching on indexing expressions.

17.2.4 Geometric recurrences

Geometric recurrences of the template show below often appear in the implementations of Fast Fourier Transforms. Unfortunately, the polyhedral model cannot directly deal with such non-linear iteration spaces.

```
for (i = 1; i <= N; i = 2*i) {  
    S;  
}
```

One possible solution is the following. First, we have to detect the presence of such recurrences within the code. The algorithm [PCS05] used in our implementation can be extended to deal with such instances. Secondly, after geometric recurrences are identified, we can rewrite the loop into a predicated form, such as the following:

```
for (i = 1; i <= N; i++) {  
    if (i is a power of 2) {  
        S;  
    }  
}
```

The predicate is still non-linear and thus has to be dealt with using approximations within the mapper. The iteration space, however, is properly enlarged and is of a form analyzable by our mapper.

17.2.5 Modulo recurrences

Another type of non-linear recurrences that happen quite often in existing applications are modulo recurrences and array indexing expressions that involve modulo arithmetic. Such recurrences appear in some Discrete Fourier Transform implementations and code that manipulates bounded circular buffers.

For example, the following program fragment involves a modulo computation on k .

```
k = 0;  
for (i = 0; i < N; i++) {  
    S;  
    k++;  
    if (k >= M) k = 0;  
}
```

It can be transformed into the following more apparent form using the same recurrence detection algorithm:

```
for (i = 0; i < N; i++) {  
    k = i % M;  
    S;  
}
```

Modulo indexing expressions in arrays may be treated in various ways. For example, if an array is used as a circular buffer, it may be array expanded into unbounded buffer to remove dependencies, and array contracted back to a circular buffer after optimizations.

18 Lowering: From Polyhedral Form to Target Code

This section describes the process in which R-Stream generates the target output. This process consists of the following subtasks:

- Convert the internal polyhedral representation to a conventional loop representation via polyhedral scanning (see Section 15 for details.)
- Convert abstract operations such as DMA initiation, thread creation, synchronization etc. into concrete operations supported by the target API. These are discussed in Sections 19, 20 and 21.
- Optimize the generated code further by running other scalar optimizations and transformations in the Sprig IR.
- Finally, perform *syntax reconstruction* to convert the internal IR back to a source level syntactic form.

18.1 Syntax reconstruction

Syntax reconstruction is the process of transforming the internal IR (see Section 16) into the surface syntax of the target language. Due to space limitations, we shall discuss the problem of syntax recovery for the C language only, although the process is analogous for other imperative languages.

The need for syntax reconstruction is driven by a few common causes:

1. We would like the output to be idiomatic, so that it can aid debugging and user comprehension. For example, early versions of R-Stream generated output in a form that resembles portable assembly language. While the output can be compiled and executed by a low level compiler, it cannot be easily understood by a human being — even an R-Stream developer. It also makes it impossible for a developer to fine tune the output code.
2. Secondly, we have discovered that many low level compilers fail to optimize a program when the output deviates from some common idioms. For example, an early version of IBM XLC compiler fails to SIMDize the following code fragment

```
k = 0;
while (k < 256) {
    A[i][j] += B[i][k] * C[k][j];
    k++;
}
```

while the semantically identical

```

for (k = 0; k < 256; k++) {
    A[i][j] += B[i][k] * C[k][j];
}

```

is properly SIMDized. This results in 2 orders of magnitude in performance difference on the CELL architecture.

18.2 Algorithm

The first step of syntax reconstruction is to recover the high level control structure of the program. We use an algorithm derived from the work of Cifuentes [Cif93, Cif94, Cif96]. The algorithm first performs a structural decomposition of the program control flow graph by identifying dominators, postdominators and loop structures.

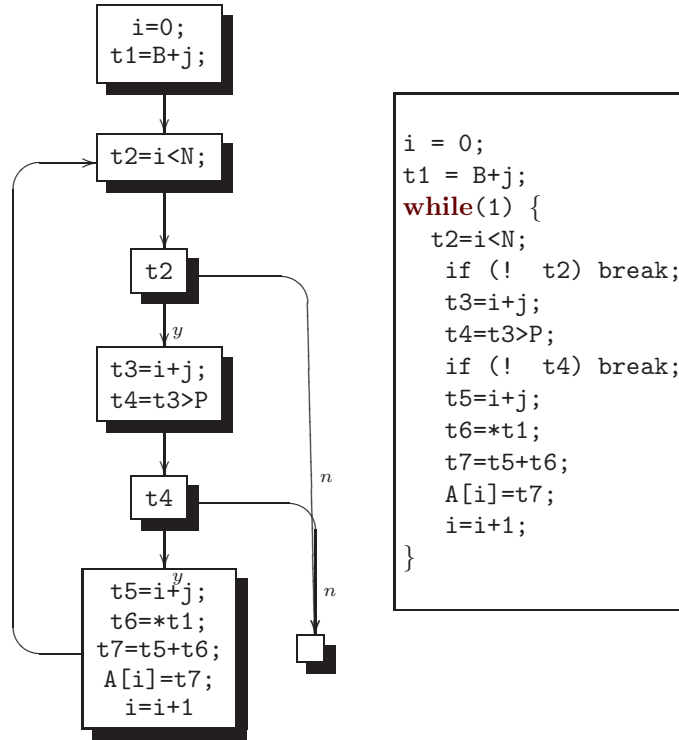


Figure 12: Control structure reconstruction.

Figure 12 illustrates the process of transforming a sample flowgraph into a C program with `while` loop. Note that because predicates in a while-loop must be C expressions rather than arbitrary C statements, the exit conditions of the loop are controlled by a sequence of `if` statements and `break`.

The next step of the syntax reconstruction process is to resynthesize larger expressions from primitive operators or simple statements containing only one operator. This process first classifies the potential side-effects of each subexpressions and whether their resulting values are shared. Side-effect free expressions and expressions whose results are used only once may be combined with other expressions without restrictions. Side-effecting expressions may also be combined with other expressions, if the side-effect can be combined with others without changing the semantics. Expressions with shared results are usually not combined with others to preserve the sharing. Occasionally, this rule can be violated if we can determine the expression is inexpensive to recompute and code duplication can make the target code more readable.

<pre> i = 0; t1 = B+j; while(1) { t2=i<N; if (! t2) break; t3=i+j; t4=t3>P; if (! t4) break; t5=i+j; t6=*t1; t7=t5+t6; A[i]=t7; i=i+1; } </pre>	<pre> i = 0; while(1) { if (i >= N i+j <= P) break; A[i]=i+j+B[j]; i=i+1; } </pre>
---	---

Figure 13: Expression building.

Figure 13 shows the result of applying our expression reconstruction algorithm to the running example. Note that the complex exit conditions can be combined into a short-circuited `||` expression. Also, the loop invariant expression `B+j` is sunk into the body of the loop and combined with the dereference operator `*` into `B[j]`, since it makes the output easier to read.²⁴

After expression reconstruction, we apply a few common idiom matching rules to further simplify the output program. This process is ad hoc in nature and highly dependent on the target language, and in some instances, on the selection of the target compiler. Figure 3 lists some of the common idioms that we match against in our current syntax reconstruction module.

Figure 14 shows the result of applying the idiom matching and rewriting

²⁴All normal C compilers can hoist the address computation out of the loop without expensive analysis if it is desired, so this transformation should not affect the performance of the target code.

Pattern	Target
<pre> i=i+1; i=i+n; while (1) { if (p) break; S } i = e; while (p) { S; i++; } </pre>	<pre> i++; i+=n; while (!p) { S } for (i = e; p; i++) { S; } </pre>

Table 3: Idiom matching.

<pre> i = 0; while(1) { if (i >= N i+j <= P) break; A[i]=i+j+B[j]; i=i+1; } </pre>	<pre> for (i = 0; i < N && i+j > P; i++) { A[i]=i+j+B[j]; } </pre>
---	--

Figure 14: Idiom matching example.

rules on our running example. The result of the transformation is an idiomatic C `for`-loop.

19 CELL Backend

The CELL Architecture is a high performance power-efficient heterogeneous chip multiprocessor developed by IBM, Sony and Toshiba as a target for graphics, cryptography, and scientific workloads. A CELL die contains one PowerPC Unit (PPU) together with eight 128-bit single-instruction multiple-data (SIMD) Synergistic Processor Units (SPU). Each SIMD can process 4 32-bit integers, single-precision floating point, or 2 double-precision float point operations in a cycle. Integer and single-precision operations are fully pipelined.

19.1 Local memory and DMA

The CELL is a distributed memory, cache-free architecture. The SPUs and PPU are connected by a high speed bus capable of peak 128 bytes/per cycle DMA bandwidth. The local memory of each SPU contains 256KB of (combined data/code) memory, with a 16 bytes/per cycle load store bandwidth, and up to 128 byte instruction prefetch per cycle. Load and stores to this memory are quadword aligned only.

One DMA controller (called the "Memory Flow Controller" (MFC)) is attached to each SPU, each containing two command queues. One queue is dedicated to the attaching SPU and the other is a "proxy" queue for receiving commands issued from the PPU. DMA operations can be initiated from the SPUs or the PPU. However, SPU initiated DMA operations are preferred, since each SPU can enqueue up to 16 requests at a time, while the PPU can only enqueue up to 8.

Each DMA request can be composed of

- one single transfer element, or
- a DMA list of up to 2K transfer elements.

Each transfer element describes the low order (32-bit) starting address and the byte count. Each element describe transfer of

- 1, 2, 4, 8, 16 bytes, with each element naturally aligned. and the source and target address must have the same 4 lowest bit, or
- multiples of 16 bytes at 16 byte alignment, up to 16KB. So, for example, it is impossible to transfer data of size 18 bytes in one DMA element.
- When transferring 128 bytes or more, the data should be aligned at 128 bytes for performance.
- The preferred transfer size is 128 bytes.

19.2 CELL target API

The R-Stream compiler currently maps to the following low level API.

Spawning executing kernels on SPUs are accomplished with the routines `CELL_mapped_begin` and `CELL_mapped_end`.

```
CELL_mapped_region_t CELL_mapped_begin(
    int region_id,
    int num_spus,
    int num_barriers,
    spe_program_handle_t * proc,
    const void * context,
    size_t context_size);
void CELL_mapped_end(CELL_mapped_region_t *);
```

The routine `CELL_mapped_begin` spawns a number of threads each running an identical copy of SPU program determined by the handle. This complementary routine `CELL_mapped_end` blocks until all spawned threads have completed. These functions can only be called from the PPU. Internally, these functions are implemented in terms of POSIX *pthreads*.

A typical example use of these routines are as follows:

```
typedef union {
    struct
        int id; /* thread id */
        double * A, * B, * C;
        s;
        int padding[16];
    } context_t;

static context_t contexts[NUM_SPUS] __attribute__((align(16)));
for (i = 0; i < NUM_SPUS; i++) {
    contexts[i].s.id = id;
    contexts[i].s.A = A;
    contexts[i].s.B = B;
    contexts[i].s.C = C;
    ...
}

CELL_mapped_region_t * R =
    CELL_mapped_begin(NUM_SPUS, 0, my_kernel,
                      contexts, sizeof(context_t));
CELL_mapped_end(R);
```

19.3 DMA primitives

Our API only supports asynchronous DMA primitives. Currently, all DMA primitives must also be initiated from the SPU side. The `CELL_dma_get` prim-

itive retrieves data into the local memory, while the `CELL_dma_put` call sends data from the local memory. The DMA calls simulate strided access when necessary by either generating a CELL DMA list dynamically, or by splitting one DMA operation into many, or by doing both. These routines also hide all alignment and size requirements violations.

```
void CELL_dma_get(
    const volatile * srcAddr,
    volatile       * dstAddr,
    size_t          bytes,
    ssize_t         srcStride,
    ssize_t         dstStride,
    size_t          count,
    int             tag);

void CELL_dma_put(
    const volatile * srcAddr,
    volatile       * dstAddr,
    size_t          bytes,
    ssize_t         srcStride,
    ssize_t         dstStride,
    size_t          count,
    int             tag);

void CELL_dma_wait(int tag);
```

All `CELL_dma_get` and `CELL_dma_put` operations are associated with a tag between 0 to 15. The `CELL_dma_wait` routine blocks until all the outstanding DMA operations with the given tag have completed.

19.4 Memory primitives

The `CELL_sync` call can only be executed in the PPU side. It is a memory barrier.

```
void CELL_sync();
```

This function should be called after `CELL_rpc` before accessing any data that may be altered by the kernel.

19.5 Synchronization primitives

The only synchronization primitive currently implemented as a barrier synchronization call:

```
void CELL_barrier(int id);
```

Each barrier call is identified by a unique identifier. The semantics of a barrier is to block the calling thread until all group members have called the same function. After all group members have entered the call, all the members are released.

Barrier objects are reserved by the runtime system in the call to `CELL_mapped_begin`.

19.6 Cell mapping example

In this section we shall demonstrate the CELL API using a blocked matrix multiply as an example. The input to the R-Stream compiler is a 1024×1024 matrix multiply:

```
float A[1024][1024];
float B[1024][1024];
float C[1024][1024];
for (i = 0; i < 1024; i++) {
    for (j = 0; j < 1024; j++) {
        for (k = 0; k < 1024; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

The output from the R-Stream compiler consists of a program to be run on the SPU and one on the PPU. On the SPU side, the function `__kernel` is generated. Within it, the 1024×1024 matrix multiply is partitioned into $32 \times 64 \times 64$ pipelined matrix multiplies, divided evenly between 8 SPUs. The special formal parameter `PROC0` refers to the current SPU number. Within `__kernel`, R-Stream allocates 2 buffers for each array A, B, and C. Each of these local arrays are of size 64×64 only.

```
void __kernel(float (A*)[1024],
              float (B*)[1024],
              float (C*)[1024],
              int PROC0) {
    float A_l_buf1[64][64], A_l_buf2[64][64];
    float B_l_buf1[64][64], B_l_buf2[64][64];
    float C_l_buf1[64][64], C_l_buf2[64][64];
    float (*A_l_v1)[64] = A_l_buf1;
    float (*A_l_v2)[64] = A_l_buf2;
    float (*B_l_v1)[64] = B_l_buf1;
    float (*B_l_v2)[64] = B_l_buf2;
    float (*C_l_v1)[64] = C_l_buf1;
```

```

float (* C_l_v2)[64] = C_l_buf2;

for (i = 0; i <= 1; i++) {
    for (j = 0; j <= 15; j++) {
        if (i == 0) {
            initialization code
        }
        Pipelined 64x64 matrix multiply
    }
}
}

```

If we expand out the code fragment titled “pipelined 64x64 matrix multiply,” the following inner pipelined loop nest appears. The loop nest executes as a 18 stage pipeline, with one prologue stage and one epilogue stage. Within the pipeline, data from the next iteration are prefetched via DMA into the reserved buffers, while the innermost 64×64 matrix multiply kernel is working on the current set of buffers.

```

// 16 stages + 1 prologue and 1 epilogue
for (k = -1; k <= 16; k++) {
    if (k <= 15 && k >= 0) {
        // Block until the prefetched data is ready
        CELL_dma_wait(0);
        swap C_l_v1 and C_l_v2,
            A_l_v1 and A_l_v2,
            B_l_v1 and B_l_v2;
    }
    if (k <= 14) {
        for (l = 0; l <= 63; l++) // Prefetch A, B and C
            CELL_dma_get(&B[64*j+1][64+64*k],
                &B_l_v2[l][0], 64*4,4,4,1,0);
        for (l = 0; l <= 63; l++)
            CELL_dma_get(&A[512*i+1+64*PROCO][64*j],
                &A_l_v2[l][0], 64*4,4,4,1,0);
        for (l = 0; l <= 63; l++)
            CELL_dma_get(&C[512*i+1+64*PROCO][64+64*k],
                &C_l_v2[l][0], 64*4,4,4,1,0);
    }
    if (k <= 15 && k >= 0) { // 64x64 matrix multiply kernel
        for (l = 0; l <= 63; l++)
            for (m = 0; m <= 63; m++)
                for (n = 0; n <= 63; n++)
                    C_l_v1[l][m] += B_l_v1[n][m] * A_l_v1[l][n];
    }
    // Block until the previous write completes
}

```

```

if (k >= 1) CELL_dma_wait(1);
if (k <= 15 && k >= 0) { // Initiate write back to C
    for (l = 0; l <= 63; l++)
        CELL_dma_put(&C_l_v1[l][0],
                     &C[512*i+1+64*PROC0][64*k], 64*4, 4, 4, 1, 1);
}
}

```

R-Stream lowering phase emits the following glue code to interface between the SPU code and the PPU code. First, a *context* structure definition is emitted. The context contains all the addresses of the matrices which have to be passed to the SPU from the PPU.

```

union __context {
    struct {
        float (*A)[1024];
        float (*B)[1024];
        float (*C)[1024];
    } context;
    double padding[2];
}

```

On the PPU side, we use the following code fragment to spawn 8 SPU threads each running the above SPU program.

```

union __context context;
extern spe_program_handle matmult1024_spu;
struct CELL_mapped_region* region;
context.context.A = A;
context.context.B = B;
context.context.C = C;
region = CELL_mapped_begin(0, 8, 0,
    &matmult1024_spu, &context, sizeof(context));
CELL_mapped_end(region);

```

On the SPU side, the main program generated. The program calls the function `CELL_spu_init` to decode the parameter `argp` and retrieve the current processor number `PROC0`. Then it reads the initial message from the PPU containing the context, then dispatch immediately to execute the `__kernel` function.

```

int main(uint64_t id, uint64_t argp)
{
    union __context c;
    uint64_t t1;
    int PROC0;

```



```

CELL_spu_init(id, argp, &PROCO);
CELL_dma_get((void *)_t1, &c,
             sizeof(c), 0, 0, 1, 0);
CELL_dma_wait(0);
__kernel(c.context.A,
         c.context.B,
         c.context.C,
         PROCO);
return 0;
}

```

19.6.1 Manual SIMDization

The XLC compiler fails to SIMDize the following inner loop nests within the function `__kernel`, leading to abysmal performance.

```

for (l = 0; l <= 63; l++)
  for (m = 0; m <= 63; m++)
    for (n = 0; n <= 63; n++)
      C_l_v1[l][m] += B_l_v1[n][m] * A_l_v1[l][n];

```

In order to gauge how R-Stream performs under more favorable conditions, the innermost 64×64 matrix multiply loop nests have been manually SIMDized. The result of the manual transformation is shown below:

```

const vector unsigned char pat0 =
  VEC_LITERAL(vector unsigned char,
    0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3);
const vector unsigned char pat1 =
  VEC_LITERAL(vector unsigned char,
    4,5,6,7,4,5,6,7,4,5,6,7,4,5,6,7);
const vector unsigned char pat2 =
  VEC_LITERAL(vector unsigned char,
    8,9,10,11,8,9,10,11,8,9,10,11,8,9,10,11);
const vector unsigned char pat3 =
  VEC_LITERAL(vector unsigned char,
    12,13,14,15,12,13,14,15,12,13,14,15,12,13,14,15);
const vector float (* restrict A_v)[64/4] =
  (const vector float (* restrict) [64/4]) A;
const vector float (* restrict B_v)[64/4] =
  (const vector float (* restrict) [64/4]) B;
vector float (* restrict C_v)[64/4] =
  (vector float (* restrict) [64/4]) C;
for (int i = 0; i < 64; i++) {
  for (int j = 0; j < 64/4; j += 4) {

```

```

// Do C[i][4*j ... 4*j+15] in parallel
// Unroll and jam
vector float C_ij0 = C_v[i][j+0];
vector float C_ij1 = C_v[i][j+1];
vector float C_ij2 = C_v[i][j+2];
vector float C_ij3 = C_v[i][j+3];
for (int k = 0; k < 64/4; k++) {
    vector float t = A_v[i][k];
    vector float a0 = spu_shuffle(t, t, pat0);
    vector float a1 = spu_shuffle(t, t, pat1);
    vector float a2 = spu_shuffle(t, t, pat2);
    vector float a3 = spu_shuffle(t, t, pat3);
    C_ij0 = spu_madd(a0, B_v[4*k+0][j+0], C_ij0);
    C_ij0 = spu_madd(a1, B_v[4*k+1][j+0], C_ij0);
    C_ij0 = spu_madd(a2, B_v[4*k+2][j+0], C_ij0);
    C_ij0 = spu_madd(a3, B_v[4*k+3][j+0], C_ij0);
    ... // etc
    C_ij3 = spu_madd(a0, B_v[4*k+0][j+3], C_ij3);
    C_ij3 = spu_madd(a1, B_v[4*k+1][j+3], C_ij3);
    C_ij3 = spu_madd(a2, B_v[4*k+2][j+3], C_ij3);
    C_ij3 = spu_madd(a3, B_v[4*k+3][j+3], C_ij3);
}
C_v[i][j+0] = C_ij0;
C_v[i][j+1] = C_ij1;
C_v[i][j+2] = C_ij2;
C_v[i][j+3] = C_ij3;
}
}

```

Trials	SIMDized	Pipelined	Time	GFlops/SPU/s
64	compiler	n	14.6s	1.46
64	compiler	y	13.9s	1.53
64	manual	n	2.273s	9.38
64	manual	y	1.658s	12.87

Table 4: Performance of matrix multiply on PS3.

Table 4 shows the performance of the manually mapped matrix multiply routines running on a PlayStation 3²⁵. As a comparison, we have also shown the performance of compiler SIMDized code and mappings with double buffering

²⁵The PS3 contains only 6 programmer accessible SPUs instead of 8.

disabled. The result shows that with manual SIMDization and double buffering, we can obtain one half of the theoretical peak performance.²⁶ The gap is most likely to be caused by remaining compiler inefficiencies in optimizing the manually SIMDized code.

²⁶Theoretical peak performance is 25.6 GFlops.

20 TRIPS Backend

The TRIPS [BKM04, SNG⁺06] backend in the R-Stream compiler is very similar to the CELL backend described in Section 19. The main differences are due to differences in the CELL and TRIPS architectures and the runtime systems.

- On the TRIPS system, processor tile 0 serves as the master processor and executes the sequential program. When a parallelized mapped region is encountered, processors 1-3 are also activated, and processor 0 switches role from a master processor to a “remote” processor. After the execution of the mapped region is completed, processor 0 switches its role again as the master processor and continues executing the unmapped part of the application program.
- The R-Stream compiler has to produce special code to allocate local memory for processor tiles on the “stream register file,” a dedicated area for fast local accesses of memory.
- The current TRIPS runtime system requires all data which have to be memory mapped onto a global shared memory segment. The R-Stream compiler identifies variables used in such a manner and converts them into offsets into the shared memory segment.

20.1 TRIPS target API

The TRIPS target API consists of the following set of system calls. This set of system calls is built on top of the TRIPS mailbox library, TRIPS DMA library and the TRIPS shared memory (`mmap`) system calls. These system calls have very similar semantics as those described in Section 19.2.

We use the calls `TRIPS_region_begin` and `TRIPS_region_end` to spawn a set of parallelized kernels.

```
TRIPS_mapped_region_t * TRIPS_region_begin(  
    int region_id,  
    int num_processors,  
    int num_barriers,  
    void (*kernel)(int proc_id, void * argp),  
    const void * context,  
    size_t context_size);  
void TRIPS_region_end(TRIPS_mapped_region_t *);
```

DMA operations are accomplished via the calls to `TRIPS_dma_get`, `TRIPS_dma_put` and `TRIPS_dma_wait`. Similar to the CELL versions of these calls, we allow multiple DMA operations to be enqueued and identified by a unique integer tag. The operation `TRIPS_dma_wait` waits until all previously issued operations have completed.

```

void TRIPS_dma_get(
    const volatile * srcAddr,
    volatile        * dstAddr,
    size_t          bytes,
    ssize_t         srcStride,
    ssize_t         dstStride,
    size_t          count,
    int             tag);

void TRIPS_dma_put(
    const volatile * srcAddr,
    volatile        * dstAddr,
    size_t          bytes,
    ssize_t         srcStride,
    ssize_t         dstStride,
    size_t          count,
    int             tag);

void TRIPS_dma_wait(int tag);

```

To allocate local memory and global memory (on a shared memory segment) we use the following system calls.

```

void * TRIPS_get_shared_memory(size_t offset);
void * TRIPS_get_local_memory(size_t offset);

```

The call `TRIPS_get_local_memory(o)` obtains the virtual address to offset *o* of the stream register file of the current processor. Similarly, the call `TRIPS_get_shared_memory(o)` obtains the virtual address to offset *o* of a global shared memory segment. Currently, DMA accessible data from the main memory must be allocated onto this segment.

20.2 Mapping example

We will now show how our favorite matrix multiply loop nests can be mapped onto the TRIPS API. For example, suppose we start with the following matrix multiply fragment.

```

double A[128][128];
double B[128][128];
double C[128][128];

...
for (int i = 0; i <= 127; i++) {
    for (int j = 0; j <= 127; j++) {

```

```

        C[i][j] = 0;
        for (int k = 0; k <= 127; k++) {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
...

```

The mapping strategy we will adopt is to take the entire **k**-loop as a kernel to be executed automatically and distribute the kernels onto 4 processors. Note that this is not meant to be an efficient mapping. For efficiency, we may want to perform iteration space tiling on the loop nests to improve locality and decrease the amount of communication. But doing so will complicate this example and detract from the subsequent exposition.

After parallelization, each processor will execute a SPMD loop that looks like the following:

```

double A[128][128];
double B[128][128];
double C[128][128];

void kernel(int proc_id, ...) {
    // Control here
    for (int i = proc_id; i <= 127; i+=4) {
        for (int j = 0; j <= 127; j++) {
            // Kernel here.
            C[i][j] = 0;
            for (int k = 0; k <= 127; k++) {
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
        }
    }
}

```

The main thread executes the kernel by spawning 4 threads running on 4 processors via the call:

```

int main() {
    ...
    TRIPS_mapped_end(TRIPS_mapped_begin(0, 4, 0, kernel, ...));
    ...
}

```

Of course, the previous mapping assumes we have shared memory available. We can make the communication explicit by inserting DMA operations around the kernel (k-loop). We obtain:

```

// global arrays
double C[128][128];
double A[128][128];
double B[128][128];

void kernel(int proc_id, ...) {
    // local arrays
    double C_loc;
    double A_loc[128];
    double B_loc[128];

    for (int i = proc_id; i <= 127; i+= 4) {
        for (int j = 0; j <= 127; j++) {
            TRIPS_dma_get(&A[i][0], &A_loc[0], 1024, 8, 8, 1, 0);
            TRIPS_dma_get(&B[0][j], &B_loc[0], 8, 1024, 8, 128, 0);
            TRIPS_dma_wait(0);
            C_loc = 0;
            for (int k = 0; k <= 127; k++) {
                C_loc = C_loc + A_loc[k] * B_loc[k];
            }
            TRIPS_dma_put(&C_loc, &C[i][j], 8, 8, 8, 1, 1);
            TRIPS_dma_wait(1);
        }
    }
}

```

The initial context of each kernel function running on the slave processors must be transferred from the master processor. We do this by packaging all the needed data in a context structure and pass a pointer to this structure to the slave processor. The first thing that the kernel function does is transfer the data from the remote copy of context into a local context variable (via DMA) so that it can be quickly accessed:²⁷

```

typedef struct {
    double (*A)[128];
    double (*B)[128];
    double (*C)[128];
} context_t;

void kernel(int proc_id, void * arg) {
    // global arrays
    double (*C)[128];
    double (*A)[128];

```

²⁷Alternatively, we can also use directly access the remote context structure because shared memory is available on the TRIPS architecture.

```

double (*B)[128];

context_t context;

// Transfer the context over
TRIPS_dma_get(arg, &context, sizeof(context), 0, 0, 1, 0);
TRIPS_dma_wait(0);

A = context.A;
B = context.B;
C = context.C;

// Rest of the kernel
}

```

The setup code on the master processor becomes:

```

context_t context;
context.A = A;
context.B = B;
context.C = C;

TRIPS_mapped_begin(0, 4, 0, kernel, &context, sizeof(context));

```

For efficiency, we also want to hide the latency of communication by software pipelining the DMA and computation code.

First, we split each local variable into two buffers, and allocate a pointer for each buffer.

```

double A_loc_buf1[128];
double A_loc_buf2[128];
double B_loc_buf1[128];
double B_loc_buf2[128];
double C_loc_buf1;
double C_loc_buf2;
double* A_loc_ptr1;
double* A_loc_ptr2;
double* B_loc_ptr1;
double* B_loc_ptr2;
double* C_loc_ptr1;
double* C_loc_ptr2;

A_loc_ptr1 = A_loc_buf1;
A_loc_ptr2 = A_loc_buf2;

```



```

C_loc_ptr1 = &C_loc_buf1;
C_loc_ptr2 = &C_loc_buf2;
B_loc_ptr1 = B_loc_buf1;
B_loc_ptr2 = B_loc_buf2;

```

Next, a macro is generated to swap the buffer pointers:

```

#define TRIPS_swap() {
    double* __A_loc_tmp;
    __A_loc_tmp = A_loc_ptr1;
    A_loc_ptr1 = A_loc_ptr2;
    A_loc_ptr2 = __A_loc_tmp;
    double* __C_loc_tmp;
    __C_loc_tmp = C_loc_ptr1;
    C_loc_ptr1 = C_loc_ptr2;
    C_loc_ptr2 = __C_loc_tmp;
    double* __B_loc_tmp;
    __B_loc_tmp = B_loc_ptr1;
    B_loc_ptr1 = B_loc_ptr2;
    B_loc_ptr2 = __B_loc_tmp;
}

```

The kernel code can then be pipelined by performing loop shifting on the DMA operations. The resulting code prefetches the data needed for the next j -iteration while operating on the current one. Two DMA tags are used to distinguish between the two groups of DMA operations. Tag 0 is for read and tag 1 is for write.

```

for (int i = proc_id; i <= 127; i += 4) {
    // prologue
    TRIPS_dma_get(&A[i][0], &A_loc_ptr2[0], 1024, 8, 8, 1, 0);
    TRIPS_dma_get(&B[0][0], &B_loc_ptr2[0], 8, 1024, 8, 128, 0);
    TRIPS_dma_wait(0);
    TRIPS_swap();
    TRIPS_dma_get(&A[i][0], &A_loc_ptr2[0], 1024, 8, 8, 1, 0);
    TRIPS_dma_get(&B[0][1], &B_loc_ptr2[0], 8, 1024, 8, 128, 0);
    *(C_loc_ptr1) = 0;
    for (int j = 0; j <= 127; j++) {
        *(C_loc_ptr1) = *(C_loc_ptr1) + A_loc_ptr1[j] * B_loc_ptr1[j];
    }
    TRIPS_dma_put(C_loc_ptr1, &C[i][0], 8, 8, 8, 1, 1);
    // steady state
    for (int j = 1; j <= 126; j++) {
        TRIPS_dma_wait(0); // wait for previous get to complete
        TRIPS_swap();
        // prefetch
        TRIPS_dma_get(&A[i][0], &A_loc_ptr2[0], 1024, 8, 8, 1, 0);
    }
}

```

```

        TRIPS_dma_get(&B[0][1 + j], &B_loc_ptr2[0], 8, 1024, 8, 128, 0);
        *(C_loc_ptr1) = 0;
        for (int k = 0; k <= 127; k++) {
            *(C_loc_ptr1) = *(C_loc_ptr1) + A_loc_ptr1[k]*B_loc_ptr1[k];
        }
        TRIPS_dma_wait(1); // wait for previous put to complete
        TRIPS_dma_put(C_loc_ptr1, &C[i][j], 8, 8, 8, 1, 1);
    }
    // epilogue
    TRIPS_dma_wait(0);
    TRIPS_swap();
    *(C_loc_ptr1) = 0;
    for (int j = 0; j <= 127; j++) {
        *(C_loc_ptr1) = *(C_loc_ptr1) + A_loc_ptr1[j] * B_loc_ptr1[j];
    }
    TRIPS_dma_wait(1);
    TRIPS_dma_put(C_loc_ptr1, &C[i][127], 8, 8, 8, 1, 1);
    TRIPS_dma_wait(1);
}
}

```

The previous transformations allocate local arrays onto the local stack of the slave processors. On TRIPS, it is preferred to map local arrays onto the stream register file to speed up accesses. This can be accomplished by replacing the previous buffer pointer allocation code with the following:

```

A_loc_ptr1 = TRIPS_get_local_memory(... offset for A1 ...);
A_loc_ptr2 = TRIPS_get_local_memory(... offset for A2 ...);
B_loc_ptr1 = TRIPS_get_local_memory(... offset for B1 ...);
B_loc_ptr2 = TRIPS_get_local_memory(... offset for B2 ...);
C_loc_ptr1 = TRIPS_get_local_memory(... offset for C1 ...);
C_loc_ptr2 = TRIPS_get_local_memory(... offset for C2 ...);

```

Finally, we must map all global data onto a shared memory segment to make them accessible to DMA. This is accomplished by replacing the initialization code running on the master processor with the following:

```

typedef struct {
    double (*A)[128];
    double (*B)[128];
    double (*C)[128];
} context_t;

typedef struct {
    double A[128][128];
    double B[128][128];
    double C[128][128];

```

```

    context_t context;
} shared_t;

int main(...) {
    // Global arrays A, B and C and context are allocated
    // in the shared segment
    shared_t shared = (shared_t *) TRIPS_get_shared_memory(0);
    double (*A)[128] = shared->A;
    double (*B)[128] = shared->B;
    double (*C)[128] = shared->C;

    ...

    // Create a context object
    shared->context.A = A;
    shared->context.B = B;
    shared->context.C = C;

    // Run the kernel in parallel
    TRIPS_mapped_end(
        TRIPS_mapped_begin(0, 4, 0, kernel, &shared->context,
            sizeof(shared->context)));
    ...
}

```

21 SMP Backend

The R-Stream compiler also contains a backend that targets Symmetric Multi-Processors (SMP). We currently use it for testing purposes, as no communications need to be generated for this target. Hence, testing the mapper on this target allows us to test everything but the communication generation and the DMA optimization components.

The SMP backend targets an API similar to the ones described in Sections 19 and 20. Among the callable routines in this API are these three:

```
// Create a new mapped region and returns its handle.
SMP_mapped_region_t * SMP_mapped_begin(
    int region_id,
    int num_threads,
    int num_barriers,
    void (*kernel)(void * arg),
    const void * context,
    size_t context_size);

// End a mapped region.
void SMP_mapped_end(SMP_mapped_region_t * region);

// Execute a global barrier.
void SMP_barrier(int id);
```

The call to `SMP_barrier` also acts as a memory barrier. The runtime system is currently implemented on top of POSIX threads, but can be easily ported to another set of threading primitives.

22 R-Stream and Polymorphous Computer Architectures

This report has focused on the R-Stream compiler, rather than the DARPA PCA program and the Morphware Forum effort within that program. Still, some brief background discussion of the relationship between PCA and R-Stream – how R-Stream serves the objectives of the PCA program – is important.

22.1 PCA program objectives

The objective of the PCA program was to develop programmable architectures which achieved high degrees of computational efficiency in *SWEPT*, that is Size, Weight Energy, Power, Time, for embedded applications of interest to the US Department of Defense. Reference applications for PCA include radar codes such as Ground Moving Target Indicator (GMTI), which has very high computational demands and which must run in demanding airborne (or space) applications where SWEPT efficiency is key. Since generally, size and weight of a computational device are largely determined by the power dissipation, we have focused on throughput (OPerations per Second, OP/S, or FLOating-point OPerations per Second, FLOP/S, the T of SWEPT) per unit of power dissipation, as the key performance metric (FLOPS/W). We call this metric the computational efficiency.

The greatest degree of computational efficiency is generally achieved with a customized chip design, an Application Specific Integrated Circuit (ASIC). An ASIC achieves high degrees of computational efficiency by being a physical embodiment of the operations and dataflow of the application on silicon. This physical embodiment allows for several optimizations, such as placing producers near consumers of data physically to reduce communication costs, and using data formats and sizes tuned to minimize switching costs. A limitation of ASICs, however, is that they are inflexible - they perform one application, and only that application. They have a slow development time, are expensive to fabricate, and small changes, as for instance bug fixes, are difficult to make. As a consequence, they are expensive and of arguable productivity.

The challenge of the PCA program was to develop computational technologies - chips, software - that could deliver performance approaching that of an ASIC, for the application classes of interest yet in a device that would still be programmable.

A further challenge in the PCA program was to provide chips that were *morphable*, that is, which could provide such efficiency over a broad class of applications. A dimension of this morphability would be to have the chip morph dynamically, that is, in the face of changing resources, mission objectives, or algorithm characteristics. One could envision, for example, resources changing with the loss or failure of computational units, or mission objective changing to favor reduced latency for a result, trading that for the precision of the result. Application characteristics might change, using the example of a radar/tracker

system, where the radar front end consists of relatively static control flow dense matrix calculations and Fourier transforms (which demands one organization or morph of the system), and the tracker back end consists of computation with a data driven control flow, as in a multiple hypothesis tracker. The PCA program worked on developing techniques to allow the architectures to morph to reoptimize performance in response to such changes.

22.2 PCA hardware strategy

The PCA program funded (in the implementation phases) four architectures for implementation: TRIPS [BKM04, SNG⁺06], Smart Memories [MPJ⁺00], RAW [TKM⁺02], and Monarch [RCCT90]. While each architecture had numerous innovations, certain common architectural features were present in all of the architectures:

- Parallelism, particularly at a coarse grain, e.g., multiple processors on the chip. The chips also provided instruction level parallelism to varying degrees.
- Distributed local memories; this in contrast to shared coherent caches, or large on-chip memories shared by all processors.
- Explicitly managed communication, in the form of DMA operations.
- Explicit synchronization among the programming elements.

More innovative features can be found in specific PCA architectures. For instance, TRIPS’s functional units expose high degrees of instruction level parallelism, Monarch provides a very efficient Field Programmable Computational Array (FPCA), RAW provides facilities such as configuring the separate processors into synchronous register coupled aggregates, and Smart Memories can reconfigure memories from being explicitly managed to being caches or translation buffers.

However, the above *common* architectural features provide, to a large degree, the basis for the computational efficiency of the chips. This is because these common features have a basis in the silicon, which is the common implementation material for the chips. Fine grained, distributed memories provide more bandwidth than centralized memories, because they provide more perimeter per unit of storage than centralized memories. Keeping computational units local to those memories reduces communication costs. Making those memories be “just memories” rather than caches, reduces the amount of energy dissipated for “chatter” traffic such as the unused part of cache lines, conflict misses, or the unnecessary write backs of data that is no longer live with respect to the algorithm. Using bulk communications allows for more efficient streaming of the communications around, onto and off of the chip, preserving precious wire and pin bandwidth for only the data that is needed for the computation.

To some extent, those principles have been known and used already, as exemplified by the presence of those features in common digital signal processing

chips, and have been for some time. Ignoring the other features of PCA chips that distinguish them, PCA architectures are still distinguished from the common DSP chips in the degree and scale, reaching over ten times (and envisioned, hundreds or thousands) their typical parallelism and granularity. PCA architectures respond, in a way, to the urgency from the recent transition where on-chip communication cost dominates computation cost, which has led to the end of the era where performance increases come from clock rate increases and architectural complexity.

It is also worth mentioning that as the PCA program has been finishing, commercial chips have emerged with similar properties. Notably, the Tiler chips are a commercialization of the RAW technology. The IBM/Sony/Toshiba CELL chip embodies similar principles, driven by the requirements of its target market of console gaming for high performance at very low power and cost. The Intel Terascale research chip provides very simple accelerators, 80 of them on a chip, each with a very small local memory. These successes validate the vision of the DARPA PCA program, and also serve as a transition and business opportunity for R-Stream, e.g., as CELL penetrates the High Performance Embedded Computing (HPEC) market for DoD computing applications, R-Stream can compile the applications.

These principles are the same as those that enable ASIC to have such good performance, but they provide programmability that ASIC's lack, by distributing control and programming through the chip. The degree of performance that can be achieved, the peak performance, is determined largely by the granularity. Finer grained hardware, lots of distributed small memories and functional units, has a higher peak performance than the equivalent silicon processor organized as a coarse grained architecture.

The challenge that PCA architectures (and their relatives, recent commercial chips) present, is in their programming. The potential of these chips is achieved *only* when the application that is running on them has a choreography of data and computation in space (over the surface of the chip, or through the system) and time that is efficient. Finding this choreography is difficult; it is parallel programming with the added responsibilities of fine grained resource management.

22.3 PCA software strategy

Significant effort in the PCA program went into developing the Morphware software architecture. The Morphware software architecture consisted of High-Level Compiler (HLC) and a Low Level Compiler (LLC). The abstraction between the HLC and LLC consisted of various virtual machines, including: Streaming Virtual Machines (SVM), Threaded Virtual Machine (TVM), and Hardware Abstraction Layer (HAL). The distinction between SVM and TVM reflects the anticipated division between code for signal processing (the static control programs, the radar) and for "data processing" (e.g., the dynamic control programs, the tracker). Abstractions, languages, were designed for describing an architecture: the machine model. Programming languages for the architectures were

investigated, e.g., stream programming languages. Different kinds and means of morphing were classified.

22.3.1 Need for definition of “mapping”

In parallel with developing the software architecture for Morphware and the various software standards, R-Stream versions 1.0 and 2.0 were being implemented by Reservoir. Our responsibilities included trying to keep track of and contributing to the various standard proposals, architecture developments, and applications implementation. As we (Reservoir) focused on implementation of the standards, we were deferring the formalization of what, exactly, “mapping” was. This deferral diminished the effectiveness of the development of the Morphware software standards. In retrospect, the need to prioritize mapping dawned on (some of) us only slowly; now it is quite apparent how central the formalisms of mapping are. For example, to decide what should go into the machine model, one should understand what kind of information the mapper needs to reshape code and optimize it for the machine. With our deferral of the formalisms of mapping, we ended up including features in the machine model language that are irrelevant to mapping, for a variety of reasons, such as: the meanings of those features were informal, or the information was something we could not (at least, with known compiler technology) incorporate into a mapping algorithm. Furthermore, in some cases, features for expressing information that a mapper would actually need, were missing in the machine model.

A similar situation occurred with the various virtual machines, where the meanings and computational models implied were not clear; how exactly to implement their semantics was unclear, etc. The meanings of the virtual machines also got snagged as the execution models were mixed up by their language bindings.

The situation also occurred with the various streaming languages. For example, Brook [Buc03] proposed language features with little insight into what was needed by a mapper: what could or could not be inferred or decided by an automatic mapping procedure; or what language features outright thwarted any attempt at automated mapping. This was aggravated by the desire to render some of the streaming languages (e.g., Brook) as improvements to C; the improvements were buffeted by the language bindings, with the meaning of the new features relative to the existing C language often unclear.

Consequently, as development proceeded with R-Stream 2.0, we found it increasingly difficult to work on bona fide mapping innovation, as more and more of our development resources were consumed trying to navigate a stormy sea of the evolving standards (which Reservoir contributed to churning). The positive feedback in this loop: lack of a mapper making it difficult to define solid standards, and the fluidity of standards making it difficult to design a mapper, reached a breaking point.

R-Stream 3.0 represents the outcome of our strategic decision (made with significant help from our DARPA program manager) to pull back from the standards to work on well-formalized bona fide mapping. This mapping would

be explicitly stated, as a well-defined mathematical optimization problem, in terms of well-described algorithm descriptions at the input, to well-defined (and simple!) execution models at the output. With those optimizations explicitly stated, we could implement (and refine) them and they would have (at least somewhat) well understood degrees of generality.

With a mapper defined in R-Stream 3.0, we are now in a position to support answering questions about the Morphware APIs, with authority. For example, any machine model feature that is proposed, can be evaluated in terms of whether or how it can enter into the mapping algorithms. Machine model attributes are with respect to a particular mapping algorithm. Similarly, a language feature in a “streaming programming language” could be evaluated in terms of whether that feature is needed by the mapping algorithm. Many features of the proposed streaming languages in PCA actually end up providing information that might be otherwise inferred by a more powerful dataflow analysis. Other features interfere with the analysis, or create “early bindings” that create more work for the mapping algorithm to undo.

22.3.2 Power and limitations of polyhedral compilers

The power of the polyhedral model, as an advance over previous approaches to high level optimization, comes from its ability to model and address a greater breadth of input program forms. When we refer to “classic” high level optimizations, we refer to optimizations that work on essentially syntactic (Abstract Syntax Tree, (AST)) representation of the program, over limited scope and types of syntax, performing incremental changes. For example, classic loop transformations are typically limited to work on single loops, or on perfect loop nests. The polyhedral form has provisions for representations of parametric imperfect loop nests. Estimates are that this encompasses the majority of certain computing benchmarks, e.g. SPEC FP [Cor] and Perfect Club [BCK⁺89].

The power of the polyhedral model for representing high level optimization has been recognized for some time; the original papers describing the model date to the early 90’s. However, using the polyhedral model was until recently considered impractical for several reasons, mainly the difficulty of lowering from the polyhedral form back into executable code. This barrier has been removed by the development of effective polyhedral scanning techniques, including those developed and implemented in R-Stream.

Another challenge of the polyhedral model is in the computational complexity of the underlying mapping stages. Generally, the computational complexity of the stages is NP-hard or worse. A barrier to the utilization of the techniques was that existing algorithms could only map kernels of code that are 10’s of lines in length. The innovations of R-Stream extend that to the 1000’s of lines. While that is still small compared to the size of a large application, it is large enough to capture interesting kernels. In most programs, the dynamic profile of the run shows that most of the cycles are in few of the lines of the program. Still, extending the size of kernels that can be mapped is a key forward research objective.

Another limitation pertains to controlling the optimizations. We have observed the situation where the amount of parallelism exposed by the mapping algorithms is so great, that the chance of finding good mappings, accounting for other concerns, is low. Tuning is needed so that the parallelization and mapping stages can “land” on good mappings more easily.

There will be more time in 2008 for us to develop a more detailed discussion of the strengths and limitations of the polyhedral model and the mapper. This will be provided in the XTRIPS final report for R-Stream, at the end of 2008.

22.3.3 Need for a uniform abstraction, phase fusion

From the start of our participation in the project, we recognized the importance of the fine grained hardware targets on our mapping progress, and this caused us to land on the *polyhedral model* as the representation of choice within R-Stream.

At a conceptual level, fine-grained, detailed management of hardware creates a challenge for automatic mapping, for compiling. This is because compilers are typically organized as a sequence of *phases* that transform or lower an algorithm to the target hardware. Each phase typically addresses some aspect of the hardware (e.g., register allocation) while leaving other aspects (e.g., the schedule of instructions) for a later phase. In principle, the ordering of the phase is arbitrary. In practice, the ordering of phases has high impact on the performance of the resulting code. This has been explored in the context of register allocation and instruction scheduling where it is possible to do one before the other. What is interesting about these ordering tradeoffs is that as the target architecture becomes more constrained, neither ordering is good, and one wants to try to do both phases together - which is in principle feasible, though it is hard to design a compiler to do so. In the running example, with a constrained register set, one should combine register allocation with instruction scheduling into a single phase.

Thus, with our high level mapper, we wanted to try to find a way to fuse phases, to be able have our mapper make fine-grained hardware tradeoffs. What the polyhedral model offers for this is a unified abstraction that represents the various optimizations of code. This creates the potential for fusing phases via the unified abstraction. While the current R-Stream 3.0 mapper has several phases, this reflects the complexity of the mapping problem. Several of the phases in R-Stream already represent unifications of steps that would otherwise be separate in a classic optimizer, e.g., the classic steps of loop interchange, skewing, reversal and others are embedded in the affine partitioning and parallelization step.

Furthermore, as we wrap up the PCA part of R-Stream’s development, we recognize that there are some new ways enabled for improving the mapping of code, e.g., ways to fuse parallelization and scheduling.

22.3.4 Power efficiency

We have not modeled power efficiency as an objective within our mapping algorithms. Rather, our mapper’s objective is to maximize throughput. The power

efficiency becomes implicit in the mapping due to the fact that the PCA architectures are intrinsically efficient. We are aware of some compiler research to optimize power explicitly, e.g., through controls provided in the API to lower clock frequency and power supply voltage for sections of code that are not time-critical. The architectures in the PCA program do not provide these hooks, but such objectives should be relatively easy to introduce into the polyhedral form mapping algorithms.

22.3.5 Dynamic Morphing

We believe that the mapping algorithms that we have developed in *R-Stream* provide insight into the way to address dynamic morphing. All along, the definition of the term “morphing” and morph dogged the participants in the Morphware program. Is “morph” a noun or a verb? When should the system morph, and how? The question is difficult because the attachment to the term “morph” entails finding a definition that suits it. A much better situation would be if there were some specific threshold that distinguishes one sort of programming or reconfiguration from another, where one would be labelled a morph and another would not. What is the difference between morphing and just being reprogrammed? Automatically doing both seem to be just facets of automatically doing mapping.

So, ultimately, the only way that questions about dynamic morphing can be answered is by specifying an algorithm. But by what principle could such an algorithm be developed? It has been our contention that a necessary preliminary step for talking about dynamic morphing is having a static mapping algorithm. The static mapping algorithm provides at least a baseline for statements about the relative performance of the online, dynamic mapping algorithm. Potentially, the static algorithm itself could form the basis of the dynamic mapping algorithm. We do not think that one can make much progress on dynamic morphing without static mapping technology.

In one regard, there is still more work to do in terms of versatility in morphs. That is, the ability to handle varied algorithm computational models, or varied execution models for the hardware. The *R-Stream* polyhedral abstraction, while highly suitable for static control programs, is not suitable (to the extent we can see) for programs with heavy data dependent control flow, e.g., multithreading programs. While some degree of approximation can allow some forms of data dependent control flow, e.g., conditionals to be rendered in the polyhedral form, more general renderings would be different.

As this report demonstrates, we are just at the threshold of achieving practical static mapping capability, via the polyhedral model. Static mapping is not easy, but should become so, soon. This technology should enable any new research efforts on dynamic mapping to have a much greater impact.

22.4 Transitions

As mentioned at the beginning of this report, **R-Stream** is available in source form to U.S. Government Departments under the Limited Rights arrangement for the DARPA contract, for Government purposes. We have delivered source distributions to AFRL, and will endeavor to provide updated source distributions, binary distributions for particular platforms, as well as support to the U.S. Government, upon request. We are interested in ports to new architectures and application domains.

We have developed a prototype license agreement for providing **R-Stream** to universities for performing academic research, and are initiating discussions with some universities. The benefit of universities working from this commercially supported source are that the polyhedral mapper is currently significantly more sophisticated than other such projects in the field (open source or closed), because it is providing fully automated transformations fully within the polyhedral representation, as well as developer interfaces for studying and improving the mapping. Also, **R-Stream** provides a robust scalar intermediate representation maintaining strict types, yet allowing faithful representation of C, and significantly powerful facilities for source idiom reconstruction. **R-Stream** also provide more robust, scalable, and powerful library features for manipulating the mathematical abstractions of the polyhedral model, than those that are currently available in the open source community.

Academic use of **R-Stream** offers a well-engineered, well-maintained basis on which to experiment, addressing quality issues inherent to the high transientness and heterogeneous programming skills of the workforce typically used in the academy. Compilers are simply too large for university research groups to stand up from scratch and maintain. Even multi-institution compiler infrastructure collaborations do not provide the needed maintenance, coherent design, solid mathematical foundations, and quality needed for a compiler to have long term impact. The presence of a profitable and profit-motivated supplier for **R-Stream** will allow university research groups to bypass the recapitulation and duplication of standing up their own technology, to work on a sound foundation, to focus on problems that are at the state of the art, with a transition path to mission and commercial use.

Finally, we have developed a plan for commercial transition developing “ports” of **R-Stream** for specific targets, or providing source licenses. We have identified more than 40 customer prospects for such commercialization activities, to begin in Summer 2008.

R-Stream has transitioned into several follow on research and development projects with the US Government. Some of these follow on the course set by the PCA program, mapping high performance computing applications to novel architectures. In DARPA Phase II SBIR W31P4Q-07-C-0144, Reservoir is enhancing **R-Stream** to produce computational code for FPGA Low-Level Compilers as computational accelerators. In Missile Defense Agency (MDA) Phase II SBIR W9113M-07-C-0072, we are extending the mapper with additional capabilities for mapping Ballistic Missile Defense (BMD) imaging applications.

We have also been using R-Stream as a base for developing additional software programming tools relevant to DOD. In Office of the Secretary of Defense (OSD) Phase I SBIR FA8650-07-M-8129, we showed how to perform software obfuscation transformations using the compiler, at both the infrastructure and the polyhedral level. In OSD Phase I SBIR W911QX-06-C-0099, we developed the capability to emit models for use in verification of software properties. In Navy Phase I SBIR N00039-08-C-0049, we are developing capabilities for interface checking for software APIs required by waveforms used in the Joint Tactical Radio System (JTRS).

22.5 Summary

The polyhedral mapping technology, as implemented in R-Stream, provides a technical foundation - precisely stated, implemented, feasible mathematical algorithms for mapping to a new class of efficient processor architectures - on which to soundly refine and transition the technologies developed in PCA.

References

- [AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 39–50, Williamsburg, VA, April 1991.
- [AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.
- [AKPW83] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.
- [AMP00a] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, pages 60–90. IEEE Computer Society, 2000.
- [AMP00b] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)* [AMP00a], pages 60–90.
- [Bas03] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'03 IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30, Ljubljana, october 2003.
- [Bas04a] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel*

- Architecture and Compilation Techniques, to appear*, Juan-les-Pins, september 2004.
- [Bas04b] C. Bastoul. *Generating Loops for Scanning Polyhedra: CLooG User's Guide*, April 2004.
- [BBK⁺07] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Affine transformation for communication minimal parallelization and locality optimization of arbitrarily-nested loop sequences. Technical Report OSU-CISRC-5/07-TR43, The Ohio State University, May 2007.
- [BC94] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–170, 1994.
- [BCC98] Denis Barthou, Albert Cohen, and Jean-Francois Collard. Maximal static expansion. In *Symposium on Principles of Programming Languages*, pages 98–106, 1998.
- [BCG⁺03a] C. Bastoul, A. Cohen, A. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, pages 209–225, College Station, october 2003.
- [BCG⁺03b] C. Bastoul, A. Cohen, A. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. Technical Report 4902, INRIA Rocquencourt, 2003.
- [BCK⁺89] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox,

- P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.
- [BDRR94] P. Boulet, A. Darté, T. Risset, and Y. Robert. (pen-)ultimate tiling ? In *Integration, The VLSI Journal*, volume 17, pages 33–51. Elsevier Science Publishers B. V., 1994.
- [Ber07] Michel Berkelaar. Lp_solve reference guide 5.5, 2007.
- [BKM04] D. Burger, S.W. Keckler, and K.S. McKinley. Scaling to the end of silicon with edge architectures. *IEEE Computer*, 37(7):44–55, July 2004.
- [BL93] T. Ball and J. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, June 1993.
- [Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [Buc03] Ian Buck. Brook specification v0.2, October 2003.
- [BW94] A. Bik and H. Wijshoff. Implementation of fourier-motzkin elimination, 1994.
- [CBF95] Jean-Francois Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 92–101, Santa Barbara, California, 1995.

- [CF93] Jean-Francois Collard and Paul Feautrier. Automatic generation of data parallel code. In Henk Sips, editor, *Proc. 4th Workshop on Compilers for Parallel Computers*, T.U. Delft, 1993.
- [Che68] N. V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.
- [Cif93] C. Cifuentes. A structuring algorithm for decompilation. In *In Proceedings of the XIX Conferencia Latinoamericana de Informatica*, pages 267–276, Buenos Aires, Argentina, 1993.
- [Cif94] Cristina Cifuentes. Structuring decompiled graphs. Technical Report FIT-TR-1994-05, Department of Computer Science, University of Tasmania, Australia, 19, 1994.
- [Cif96] Cristina Cifuentes. Structuring decompiled graphs. In *Proceedings of the International Conference on Compiler Construction (CC'96). Lecture Notes in Computer Science 1060. Linkoping, Sweden.*, pages 91–105, April 1996.
- [Cli95] Cliff Click. Global code motion/Global value numbering. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 246–257, La Jolla, California, June 1995.
- [Cor] Standard Performance Evaluation Corporation. Description of the cfp2000 benchmark.
- [CP95] C. Click and M. Paleczny. A simple graph-based intermediate representation. In *The First ACM SIGPLAN Workshop on Intermediate Representations*, San Francisco, CA, 1995.

- [CS95] Keith Cooper and Taylor Simpson. SCC-based value numbering. Technical Report CRPC-TR95636-S, Center for Research on Parallel Computation, Rice University, October 1995.
- [CSV01] Keith Cooper, Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, 23(5), September 2001.
- [DRV00] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser, 2000.
- [DSV03] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. In *ACM CASES*, pages 298–308, 2003.
- [DSV04] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. Technical Report 2004-23, École Normale Supérieure de Lyon, INRIA, 2004.
- [DSV05] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Trans. Computers*, 54(10):1242–1257, 2005.
- [dt07] The JGAP development team. JGAP documentation v3.2.2, 2007.
- [DV94] A. Darte and F. Vivien. Automatic parallelization based on multi-dimensional scheduling, 1994.
- [DV95] A. Darte and F. Vivien. Revisiting the decomposition of Karp, Miller and Winograd. *Parallel Processing Letters*, 5(4):551–562, December 1995.
- [Fea88a] P. Feautrier. Array expansion. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, 1988.

- [Fea88b] P. Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, 1988.
- [Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53 (or 23–52??), 1991.
- [Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.
- [Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part II. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.
- [Fea96] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103, 1996.
- [Fea03] Paul Feautrier. Solving systems of affine (in)equalities: Pip’s user’s guide. fourth version. Technical report, PRiSM - Laboratoire de Recherche en Informatique, October 2003.
- [FO01] Björn Franke and Michael F. P. O’Boyle. Compiler transformation of pointers to explicit array accesses in dsp applications. In *CC ’01: Proceedings of the 10th International Conference on Compiler Construction*, pages 69–85, London, UK, 2001. Springer-Verlag.
- [FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimizations. *ACM Transaction on Programming Languages and Systems*, 9(3):319–349, June 1987.

- [GFL04] M. Griebl, P. Faber, and Ch. Lengauer. Space-time mapping and tiling; a helpful combination. *Concurrency and Computation: Practice and Experience*, 16:221–246, 2004.
- [GHF⁺05] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Yukio Watanabe, and Takeshi Yamazaki. A novel simd architecture for the cell heterogeneous chip-multiprocessor. In *Hot Chips 17*, August 2005.
- [GHF⁺06] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in cell’s multicore architecture. *IEEE Micro*, March 2006.
- [GLL95] Junjie Gu, Zhiyuan Li, and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing ’95*, 1995.
- [GLW98] Martin Griebl, Christian Lengauer, and S. Wetzel. Code generation in the polytope model. In *IEEE PACT*, pages 106–111, 1998.
- [GR07] Gautam Gupta and Sanjay Rajopadhye. The Z-polyhedral model. In *PPoPP ’07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 237–248, New York, NY, USA, 2007. ACM Press.
- [Gri00] M. Griebl. On the mechanical tiling of space-time mapped loop nests. Technical Report MIP-0009, Fakultät für Mathematik und Informatik, Universität Passau, 2000.
- [hpf] High-performance fortran language specification version 1.0.
- [HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *SIGPLAN Confer-*

- ence on Programming Language Design and Implementation, pages 254–263, 2001.
- [IT88a] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages* [IT88b], pages 319–329.
- [IT88b] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 319–329, New York, NY, USA, 1988. ACM Press.
- [Kel96] Wayne Kelly. *Optimization within a Unified Transformation Framework*. PhD thesis, University of Maryland, 1996.
- [KPR95] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)*, page 332. IEEE Computer Society, 1995.
- [KPR98] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. Technical Report CS-TR-3317.1, University of Maryland, Computer Science Department, October 1998.
- [Len93] Christian Lengauer. Loop parallelization in the polytope model. In *International Conference on Concurrency Theory*, pages 398–416, 1993.
- [LeV92] H. LeVerge. A note on chernikova's algorithm, 1992.
- [LL97] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the*

Twenty-fourth Annual ACM Symposium on the Principles of Programming Languages, Paris, France, 1997.

- [LLL01] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. *ACM SIGPLAN Notices*, 36(7):103–112, 2001.
- [MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array-data flow analysis and its use in array privatization. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 2–15, New York, NY, USA, 1993. ACM Press.
- [MD05a] Dominic Mallinson and Marc DeLoura. Cell: A new platform for digital entertainment. Technical report, SCEA U.S. R & D, 2005.
<http://research.scea.com/research/html/CellGDC05/index.html>.
- [MD05b] Dominic Mallinson and Mark Deloura. CELL: A new platform for digital entertainment
<http://research.scea.com/research/html/cellgdc05/index.html>, 2005.
- [MN03] MathWorks and NIST. Jama: A java matrix package, 2003.
- [MPJ⁺00] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *Proc. of International Symposium on Computer Architecture*, pages 161–171, June 2000.
- [NR00] Sunder Phani Jumar Nookala and Tanguy Risset. A Library for Z-Polyhedral Operations. Technical Report 1330, IRISA - Institut De Recherche en Informatiq et Systèmes Aléatoires, 2000.

- [PCS05] S. Pop, A. Cohen, and G. Silber. Induction variable analysis with delayed abstractions. In *In 2005 International Conference on High Performance Embedded Architectures and Compilers*, Barcelona, Spain, 2005.
- [Pug92] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Communication of the ACM*, August 1992.
- [QR] F. Quilleré and S. Rajopadhye. On code-generation in the polyhedral model.
- [QRR96] Patrice Quinton, Sanjay Rajopadhye, and Tanguy Risset. On manipulating Z-polyhedra. Technical Report 1016, IRISA - Institut De Recherche en Informatiq et Systèmes Aléatoires, 1996.
- [QRW00] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5), October 2000.
- [RCCT90] R.D. Rettberg, W.R. Crowther, P.P. Carvey, and R.S. Tomlinson. The monarch parallel processor hardware design. *Computer*, 23:18–30, April 1990.
- [RR04] Lakshminarayanan Renganarayana and Sanjay Rajopadhye. A geometric programming framework for optimal multi-level tiling. In *High Performance Computing, Networking and Storage Conference (SC2004)*, November 2004.
- [SC04] Robert Schreiber and Darren C. Cronquist. Near-optimal allocation of local memory arrays. Technical Report HPL-2004-24, Hewlett-Packard Laboratories, February 2004.

- [Sch86] Alexandeer Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., 1986.
- [SGI07] SGI, Inc. *Reconfigurable Application-Specific Computing User's Guide*, 2007. Document number 007-4718-006 available online at <http://techpubs.sgi.com/>.
- [Sim98] Loren Taylor Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Rice University, 6, 1998.
- [SJHM06] David A. Schwartz, Randall R. Judd, William J. Harrod, and Dwight P. Manley. The vector signal image processing library 1.2 api. Technical report, HRL Laboratories, Space and Naval Warfare Systems Center, Silicon Graphs Inc./Cray Research and Compaq Computer Corp./Digital Equipment Corp., apr 2006.
- [SL99] Yonghong Song and Zhiyuan Li. A compiler framework for tiling imperfectly-nested loops. In *Languages and Compilers for Parallel Computing*, pages 185–200, 1999.
- [SL06] Rachid Seghir and Vincent Loechner. Memory optimization by counting points in integer transformations of parametric polytopes. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 74–82, New York, NY, USA, 2006. ACM Press.
- [SLM07] Rachid Seghir, Vincent Loechner, and Benoit Meister. Counting points in integer affine transformations of parametric z-polytopes, 2007.
- [SNG⁺06] K. Sankaralingam, R. Nagarajan, P. Gratz, R. Desikan, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan,

- S. Sharif, P. Shivakumar, W. Yoder, R. McDonald, S.W. Keckler, and D.C. Burger. The distributed microarchitecture of the trips prototype processor. In *39th International Symposium on Microarchitecture (MICRO)*, December 2006.
- [Tea02] The Polylib Team. Polylib user’s manual. Technical report, University of Louis Pasteur, Strasbourg, France, September 2002.
- [TKM⁺02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen and Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general purpose programs. *Micro*, Mar 2002.
- [TP01] Peng Tu and David A. Padua. Automatic array privatization. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 247–284, 2001.
- [Tu95] Peng Tu. Automatic array privatization and demand-driven symbolic analysis. Technical Report UIUCDCS-R-95-1911, University of Illinois at Urbana-Campaign, May 1995.
- [Vas07] Nicolas T. Vasilache. *Scalable Program Optimization Techniques in the Polyhedral Model*. PhD thesis, Universit Paris Sud XI, Orsay, September 2007.
- [VBC06] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC’06)*, lncs, pages 185–201, Vienna, Austria, March 2006. Springer-Verlag.

- [WCES94] D. Weise, R. Crew, M. Ernst, and B. Steensgaard. Value dependence graph: Representation without taxation. In *ACM Symposium on Principles of Programming Languages*, pages 297–310, 1994.
- [Wil93] Doran Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA - Institut De Recherche en Informatiq et Systèmes Aléatoires, December 1993.
- [WL91] Michael E. Wolf and Monica Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.
- [WL94] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. Technical Report CS-TR-1994-1248, University of Wisconsin-Madison, 1994.
- [WL02] John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Static Analysis Symposium*, September 2002.
- [Wol96] Michael Wolfe. *High Performance Compilers For Parallel Computing*. Addison Wesley, 1996.
- [WZ91] M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.
- [XHG05] J. Xue, Q. Huang, and M. Guo. Enabling loop fusion and tiling for cache performance by fixing fusion-preventing data dependences. In *International Conference on Parallel Processing (ICPP'05)*, pages 107–115, 2005.

- [Xue97] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

Index

- affine scheduling, 34
- array contraction, 95
- array expansion, 30

- blocking, *see* tiling

- data locality
 - spatial locality, 5, 41
 - temporal locality, 5, 41
- double-buffering, 74

- grouping, *see* tiling

- jamming, 94

- modular mapping
 - see* modulo mapping, 96
- modulo mapping, 96
- multi-buffering, 74

- placement, 57
- polyhedral model, 10
- profitability, 44

- raising, 117
- register tiling, 94

- strided, 41

- task, 42
- tiling
 - data tiling, 42
 - iteration tiling, 42